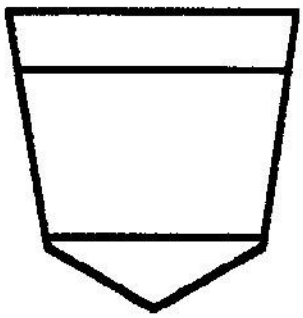
MACHINE LANGUAGE PROGRAMMING

SHARP PC-1500 & RADIO SHACK PC-2 POCKET COMPUTERS



© Copyright 1983 POCKET COMPUTER NEWSLETTER

MACHINE LANGUAGE PROGRAMMING
THE SHARP PC-1500
AND RADIO SHACK PC-2
POCKET COMPUTERS

Hidden beneath the surface of pocket computers such as the Sharp PC-1500 and Radio Shack PC-2 models is a secret world of unique and powerful capabilities. Alas, this world is only open to the initiated who have the willingness and stamina to study and learn *machine language programming!*

Machine language programming (or MLP as it will frequently be referred to here) is really what makes the entire system work. Indeed, the BASIC language itself and all the capabilities of the pocket computer are programmed in this most fundamental language of the microprocessor.

MLP is a complex subject. The reason computer languages such as BASIC were developed is because the average user of computers would like to be able to use them easily. Such people do not want to be bogged down in a myriad of technical details about how the computer itself operates. Details that have little to do with solving the problem at hand. They want to be able to quickly express to the computer how to perform a specific set of operations in order to obtain desired answers. The trade-off they make (frequently without knowing it) is one of speed of programming versus speed of computer operation. The speed of a computer, compared to that of the human mind, is vastly superior. Allowing it to take hundreds or even thousands of times longer than its most efficient arrangement would require is often unnoticeable to the casual user. Thus, the average user has little regard for the fact that the PC may take a thousand times longer to execute a program written in BASIC than it does to perform the same procedure when written in machine language.

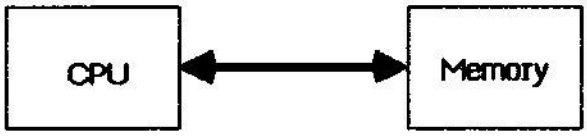
When one programs in machine language one has to be concerned with a lot of technical details. These details often have little to do with solving the overall problem for which the computer is being used. If, for example, one has to perhaps deal with 20 or 30 parameters when programming a computer using a language such as BASIC, one might have to deal with 500 to 1000 pieces of information (instructions, storage locations, sequences, procedures, etc.) in order to successfully program the same problem using machine language. The reward for being able to do this, however, might well be an overall increase in the speed at which a specific problem could be solved. This improvement in the rate at which a specific task might be accomplished can often be measured in orders of magnitude, such as ten, one hundred or a thousand of times faster!

Access to the computer at the machine level can also open a whole new world of opportunities. This is especially true if one wants to deal at the hardware level in such areas as using the PC to perform real-time control operations. Generally these kinds of specialized applications cannot be successfully approached using high level languages, such as BASIC, because of speed or physical interfacing restraints.

A Piece at a Time

As complex as machine language programming can be, it is possible to make the chore of learning how to program in machine language be relatively easy to bear. The trick lies in breaking down the operation of a computer into its simplest parts. This makes each section easier to understand. Eventually the pieces can be placed back together as one gains familiarity with the fundamental aspects.

How simple can we get? How about defining a computer as consisting of two essential divisions: (1) a CPU and (2) memory.



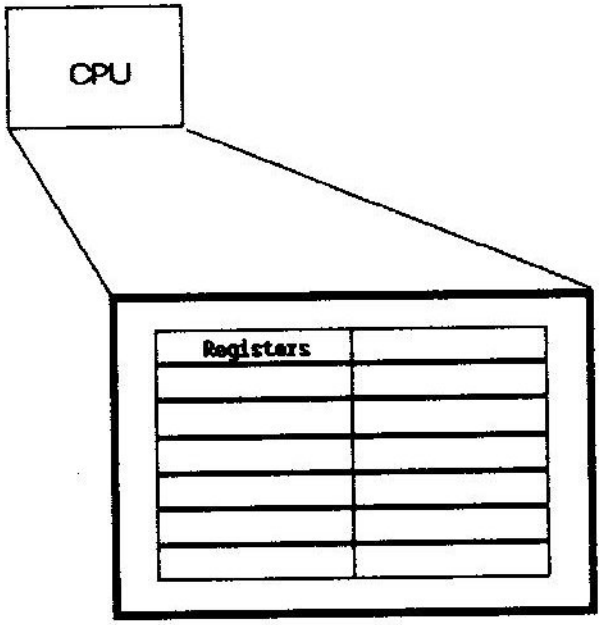
The CPU (Central Processing Unit) in a pocket computer is a microprocessor. That is, it is an integrated circuit that is the heart of the system. In the Sharp PC-1500 the CPU integrated circuit is referred to as an LH5801. It is a proprietary device made especially for that pocket computer. (Since the Radio Shack PC-2 is simply a custom-made version of the Sharp PC-1500, unless otherwise noted in this text, operation of the PC-2 can assumed to be the same.) The nomenclature LH5801 has no special significance to anyone other than the manufacturer. It is essentially nothing more than a part number to identify that particular type of electronic device.

A CPU such as the LH5801 is able to do a number of essential operations. But, basically what is does is transfer information to and from memory. Part of what it transfers is its own instructions. That is, the directives that tell it what to do!

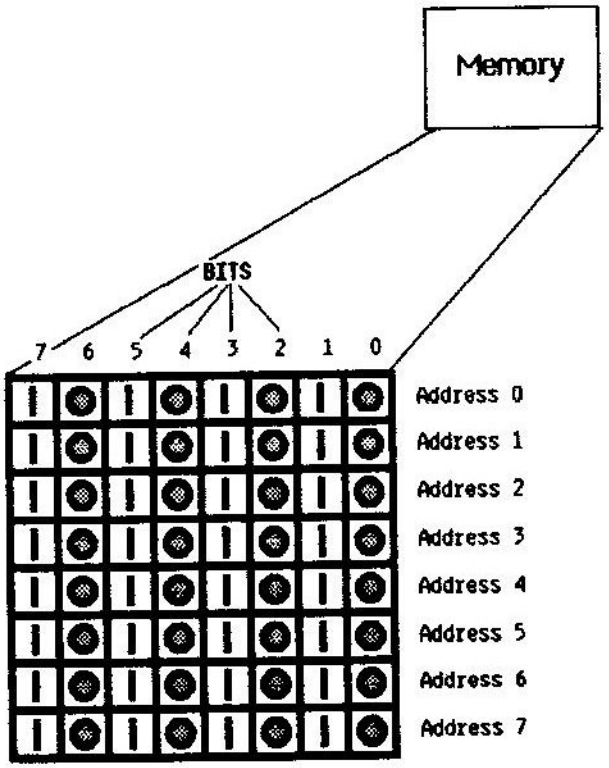
The memory elements in a computer hold two basic types of information. Instructions that tell the CPU what to do and data that is used to solve problems. To the uninitiated it is impossible to tell which parts of memory do what. You will soon become among the initiated.

A Step at a Time

A CPU essentially consists of a number of registers, some electronic paths that interconnect these registers, and an array of logic-performing circuits that control what happens within and between the registers.



Memory essentially consists of an orderly arrangement of registers and a method of addressing or selecting each register. In theory there could be up to 65,536 registers (64K) in the main memory bank of a PC-1500, each register of which would have a unique address in the range 0 - 65,535. (Because of the manner in which electronic circuits operate, numbering of items such as registers and cells within computers always start with zero.) Memory registers in a PC-2 have eight cells. Each cell can assume the binary state of one or zero. With eight cells making up a register, 256 different binary patterns or values (ranging from 0 through 255) can be stored in each memory register.

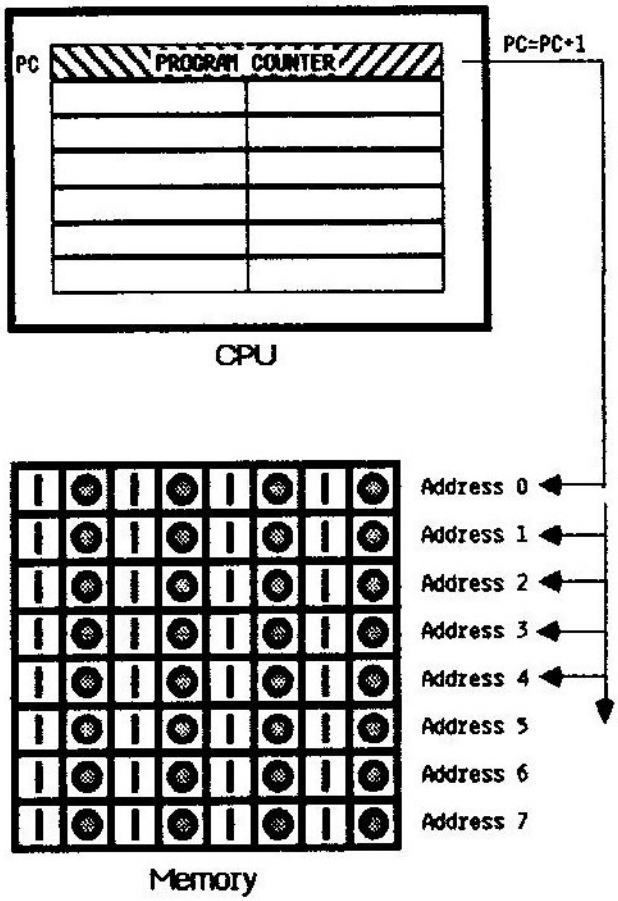


The Program Counter

The first thing that has to happen in order for the CPU to communicate with memory is for it (the CPU) to have an "address" specifying a particular memory register with which it is to transact its business. Would you believe that there is a special register within the CPU whose sole task is to keep track of where the CPU should obtain its next "communications" or instruction? Yep! And it has a very apt name: the *program counter.* The program counter in a PC-1500 is 16 bits wide. Now it just so happens that a register of 16 binary cells can represent up to 65,536 patterns -- in the range 0 through 65,535 -- which (coincidently, eh?) turns out to be the maximum number of memory registers that a pocket computer of the type being discussed can theoretically have in its primary memory banks. Thus, the program counter can in principle hold all the values that could translate into valid memory register addresses.

Whenever the CPU needs to obtain a instruction or an instruction-related piece of

information from memory, it looks at the contents of the program counter to ascertain the appropriate memory address.

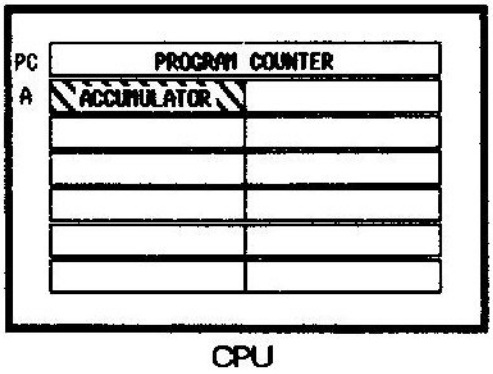


Once started, the operation of the program counter is essentially automatic. Whenever an instruction is "fetched" from a memory location, the value contained in the program counter is automatically incremented to "point" to the next memory address. There are, however, a few exceptions to this automatic incrementing sequence. A few special classes of directives can alter the contents of the program counter. Can you guess these classes? That is hardly a fair question at this point, but as you will eventually learn in detail, the classes of instructions known as "jumps" and "calls" can change the contents of the program counter.

How does a computer such as the PC-1500 get started? Well, one of the first things that happens when power is applied to the unit is that the program counter is set to contain the address in memory where the CPU is to begin finding machine language instructions!

The Accumulator

There is an 8-bit register in the CPU that can sort of be considered as a "jack-of-all-trades." This register is able to hold information while the CPU is in the process of performing other operations -- such as fetching another instruction from memory. It also works in conjunction with other internal CPU registers such as the arithmetic and logic unit (ALU). In this regards it is capable of performing mathematical operations such as addition, subtraction and Boolean logic (OR, NOT, AND, etc.). When a series of calculations are being performed this register can directly accumulate intermediate results. Hence the derivation of its name: *accumulator.*



One of the most frequent uses of the accumulator (which will frequently be abbreviated as the A register in this text) is simply to serve as a "scratch pad." That is, it is simply used to hold a number or binary pattern while the CPU obtains another operator. The accumulator in the PC-1500 can be loaded with a value obtained from memory or from another CPU register.

When directed to do so, the value in the accumulator can be added or subtracted from the value of another CPU register or a location in memory. It can also perform logic operations with the contents of other CPU registers or memory locations.

Also, certain instructions can cause the contents of the accumulator to be shifted to the right or left. This capability serves many useful purposes. It is one way in which multiplication or division can be implemented.

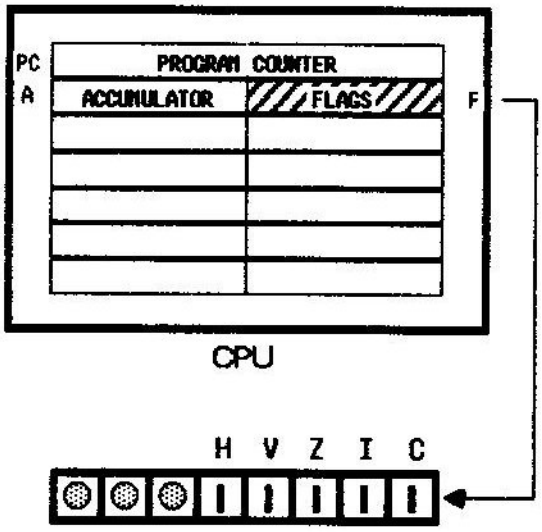
You will deal with the accumulator frequently when doing machine language programming. There will be much to learn about its versatility.

Flags

Flags are single logic cells that can assume one of two states: set or reset (the latter sometimes being referred to as "cleared"). The PC-1500 has five separate flags. While each flag operates independently, as far as the programmer is concerned they can be viewed as being grouped in the five least significant bit positions of an 8-bit *flag register.* This concept is important because at various times it will be important to be able to save or restore the status of all five flags at one time.

Special instructions enable the flag register to be manipulated as an entity to accomplish this objective.

While the flags may be viewed as residing within one register, each flag serves a specific purpose and is operated independently of the other flags. These flags are used to indicate the results of various operations following the execution of certain classes of instructions. It will eventually become necessary for a prospective machine language programmer to thoroughly understand the purposes of each flag. However, for now a brief introduction to their functions will suffice.



The *zero flag*(Z flag) is used to indicate if the contents of a register are zero (flag set) or if the contents of a register are non-zero (flag cleared). Note the seemingly inverse relationship here!

The *carry flag* (C flag) is set when there is a carry from the most significant bit of a register and cleared if there is no carry during an addition operation. During subtraction it is set if there is no borrow and cleared if there is a borrow. The carry flag can also be considered as an extension of a register during certain types of operations such as shifts and rotates. It can also be independently set or cleared by special instructions so that its status can be defined at times determined by the programmer.

The *half carry flag* (H flag) is set when there is a carry from the least significant four bits of a register during certain types of decimal arithmetic operations. It is cleared when a carry does not occur from this lower half of a byte. (The eight binary cells that make up a register are sometimes referred to as a *byte*. Did you know that half of a byte is sometimes referred to as a nibble? No joke!)

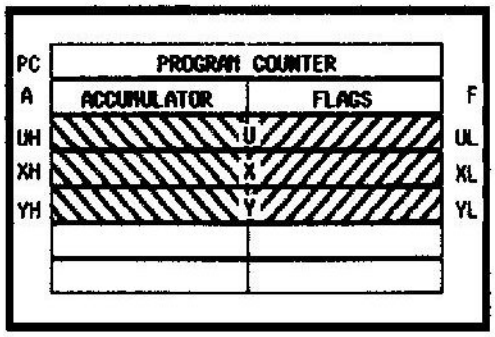
The *overflow flag*(V flag) is set or reset as a function of the carries from the most significant two bits of a register. It has value in certain mathematical procedures. Many ML programmers never have occassion to make use of this particular flag.

The *interrupt enable flag* (I flag) is used to enable or disable a type of CPU interrupt capability provided for the LH5801 CPU. Most programmers will not have to be concerned with its applications.

Data Pointers and General Purpose Registers

The LH5801 has a whole group of registers that may be used for several purposes. While not having quite the versatility of the accumulator (in that they generally lack "calculating" capabilities), never-the-less they come in very handy.

There are six of these *general purpose* 8-bit registers. However, they can be paired to form three sets of 16-bit *data pointer* registers. When serving as a data pointer the contents of the register may be used to indicate an address in memory where data is to be obtained or deposited. Note that when coupled together to form a 16-bit register they can hold any address in the range 0 - 65,535, thus they can point to any valid address in memory that the CPU is theoretically capable of accessing.



When not being used as data pointers, these registers may be used to temporarily store and manipulate information in 8 or 16-bit format. These registers are rather arbitrarily referred to as the X, Y and U register pairs when referenced as 16-bit data pointers. Since each 16-bit pair can also be manipulated as two independent 8-bit groups, each may be referred to as consisting of a high byte and a low byte. Thus the designations XH & XL, YH & YL and UH & UL when the 8-bit registers are referenced as separate entities.

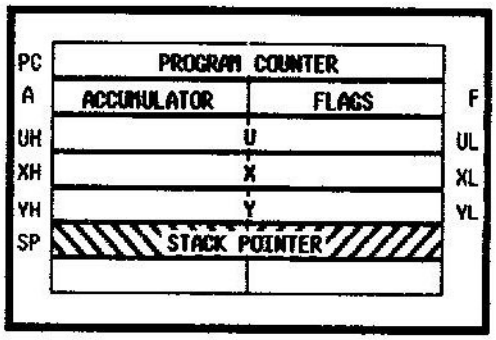
It is worth noting that while a data pointer may be the same 16-bit size as the program counter, their functions are different. Remember, the program counter always tells the CPU where to obtain the next *instruction* in memory. On the other hand, a data pointer register may be used to tell the CPU where to obtain *data* from memory. The two operations are quite distinct and should

 Copyright 1983 POCKET COMPUTER NEWSLETTER

not be confused.

The Stack Pointer

The *stack pointer* is a special 16-bit register that has a variety of interesting uses. Basically, what it does is point to a specific address in memory where information is to be stored on a temporary basis. The stack pointer register is designed such that each time it is utilized by an instruction, its contents are incremented or decremented depending on the type of instruction being performed. This operation provides a method whereby information may be *pushed* into a storage area in memory or *popped* out of that region back into designated CPU registers.



CPU

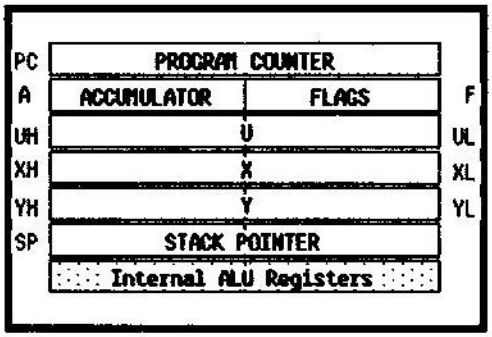
A frequent use of the stack pointer is to save the contents of the program counter (remember what it does?) when a program branches from a main sequence of instructions to a minor sequence. This process is generally known as subroutining. By saving the value of the PC in an area of memory indicated by the stack pointer, the CPU is able to eventually resume operations from the point where the subroutine was first called.

While all of this may seem complicated at this time, don't worry about the details, just grasp the concept. The details of stack pointer operation will be presented when it is appropriate to understand it in depth.

Internal (User Transparent) CPU Registers

The registers that have been discussed are all in some manner or another accessible to the programmer. That is, by giving the proper instructions or sequences of such directives, the contents of those locations can be directly controlled.

There are a few internal registers in the CPU that do a lot of work that is effectively hidden from the view of the user. For instance, there are registers that perform certain types of arithmetic operations. As a group these comprise and are generally referred to as the arithmetic logic unit (ALU for short). There are also registers that control the overall operation of the CPU. One of the most important of these is known as the IR or *instruction register.* This register holds the binary pattern representing an instruction (fetched from memory) while surrounding circuitry "decodes" the instruction. This results in the CPU performing the operation dictated.



CPU

It is often worthwhile remembering that the initial description of a computer as consisting of a CPU and memory is not inaccurate. All a computer really does is *fetch* an instruction from memory and then *execute* the directive it receives. The beauty and power of the computer comes from the fact that it can perform a typical fetch and execute cycle at the machine language level in just a few millionths of a second. That is the case even for a tiny hand-held package such as the PC-1500!

An Instruction at a Time

Just as a computer executes one instruction at a time, a potential machine language programmer should plan on learning about one instruction at a time. While the PC-1500 can execute several hundred different instructions, don't be faint-hearted. Many of the instructions are identical except for the fact that they operate on a different register. Thus, once the concepts for a fundamental class have been learned, you will know how to use a whole group or series of particular directives.

A Treatise on Mnemonics

Mnemonics (pronounced "knee-mon-ics") are memory aids. That is they are a shorthand way of writing machine language directives. They are generally developed in such a way that they help the programmer remember what each instruction does.

The novice programmer is sometimes confused about the true significance of mnemonics. It is important to realize that to a computer at the operational level, there is no such thing as a mnemonic. All the CPU ever sees as it processes instructions are the binary patterns that represent each particular directive. Remember, the CPU has an instruction "decoder" network (circuit) that

translates each different type of pattern into a sequence of events to accomplish a particular objective. Unfortunately, people do not appear to have such efficient decoder networks in their heads *when it comes to manipulating binary patterns*. However, the human mind is quite adept at handling alphanumeric patterns (probably because that is what they are trained or "programmed" to recognize). People have to program computers. Hence, we need to have an efficient method of remembering what kinds of instructions our computers can perform and of writing those directives down so that *we* can easily tell what it is we are doing.

Mnemonics are completely artificial, abstract and arbitrary definitions of machine language instructions. Anybody that wants to can create and compile their own set of mnemonics for any computer. The only reason for having any standardization when it comes to using mnemonics is so that other people with whom we converse and communicate can readily understand what it is we are talking about.

Now it happens that most CPU manufacturers get the honor of promulgating a set of mnemonics to properly represent the instruction set of their beloved CPU. After all, they want people to use their machines. To make life easier on prospective customers, they generally are happy to put forth an easy way to remember all the various machine language instructions that their CPU can perform.

Of course, there are always exceptions, aren't there? When the Sharp PC-1500 was first introduced, it seems that the manufacturer *did not want consumers to know how to program it in machine language!* They thought BASIC programming would be good enough for everybody.

Alas, they were incorrect in their reasoning. A large group of purchasers were indeed interested in utilizing the little PC at its most powerful level. Some of these users, in particular a group of *POCKET COMPUTER NEWSLETTER* readers, worked together to decipher the machine language instruction set that the PC was apparently capable of performing. This was done entirely by analyzing the binary patterns stored in the ROM (read-only memory) portions of the PC.

The principal investigator of this group, *Norlin Rober*, created a set of original mnemonics to represent the instructions that the group discovered. The mnemonics he created are the ones used in this text. They are referred to herein as the *Rober Mnemonics*.

Here is some more information that may help you to further understand the function of mnemonics. If, after you have designed a machine language program using mnemonics, you plan on having your PC execute your directives, you will need to perform a translation from the mnemonics *you* use to the binary patterns (machine code) utilized by the CPU. To do this (for small programs) you will simply use a lookup table. That table will list the mnemonics you use and the equivalent binary codes that must be fed to the CPU to represent each instruction.

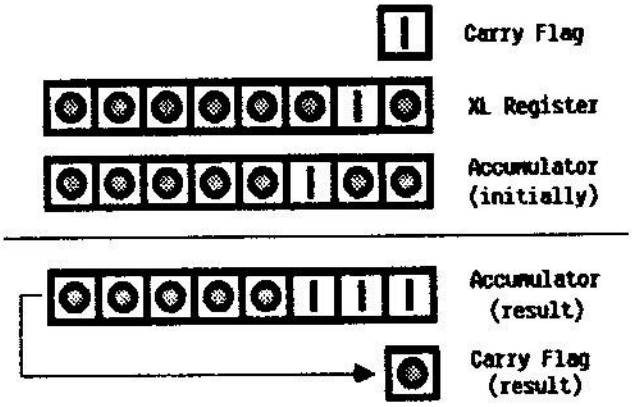
Now, if you plan on going into the "big leagues," (such as making your living programming in machine language) then you might eventually want to use a special program that would help you convert from mnemonic format to machine code. Such a program is called an *assembler*. It is nothing more than a program that performs automatic table lookup procedures! Trouble is, it in itself can take up a lot of memory and on a small computer such as a PC can be more of a pain to use than not (due to all the shuffeling of programs in and out of memory, etc.). Such being the case, next thing you know you are talking about using a "cross-assembler." That is an assembler program that works on a larger computer and produces code for the "target" (your PC-1500) machine. Before getting carried away with a discussion about such programs, let me end it by saying that the use of such programs will *not* be assumed in this text. Instead, I shall proceed on the basis that small programs, of the type beginning machine language programmers will feel confident to tackle, will be converted from mnemonics to machine code by the manual (table lookup) method!

Let's Add Some of This Up

A good way to get a feel for programming at the machine code level is to take an in-depth look at an instruction. We are going to start with a register-to-register addition directive. The machine code representation for this instruction is the 8-bit binary pattern 0 0 0 0 0 0 1 0 which can be represented in hexadecimal numeric shorthand as the value 02. The mnemonic for this command is: ADDA XL. This represents a shorthand method of saying: add the contents of register A (the accumulator) and the contents of the 8-bit register XL (the lower half of the 16-bit register X).

There are some other things you need to know about this directive: (1) The type of addition performed is based on two's complement binary arithmetic, (2) The initial contents of the carry flag (C) are added to the least significant bit position at the start of the addition process, (3) The result of the addition process is left in the accumulator (A register), (4) Any "overflow" from the addition process will be reflected in the status of the carry flag (C) at the conclusion of the operation, (5) The status of the Z, V and H flags (along with the previously mentioned C flag) may be altered by the results of the operation, and (6) The original

contents of register XL are not altered by the operation.



Gee! That is quite a list of information to have to know just to effectively utilize a single machine language instruction. All of it is important, too. If you forget just one aspect while creating a program, you can end up with a procedure that does not yield correct results. I can tell you from years of experience that one of the items many novice programmers forget about when attempting to apply this type of directive is the affect caused by the initial condition of the carry flag! Note that if the carry flag is set (at a logic one state) when this instruction is executed, that a count of one will be added to whatever is the result of the addition. This feature has considerable practical application. It provides a means of performing what is commonly known in the field as multiple-precision arithmetic. That is a method whereby arithmetic values may be represented by the combined contents of several registers. On the other hand, it can be a subtle (and vicious) trap for the programmer who forgets its ramifications when performing simple register-to-register (often referred to as single-precision) arithmetic. If the status of the carry flag is not known (i.e., reset to the zero state) prior to executing the directive, one can end up with an addition that seems to be mysteriously off by a count of one extra. The reason this operational aspect can really throw beginning programmers is because in a complex program, the status of the carry flag may be varying each time such an instruction is encountered. The net result is a program that capriciously yields incorrect mathematical results!

All of this points out one critical factor about machine language programming. It can be extremely complex and one subtle mistake on the part of a programmer can lead to errors, the cause of which are often difficult to detect. To get a better appreciation of the potential for catastrophe, contemplate the operation of a machine language program with, say, about 10,000 instructions. An error in the application of any one of those could cause incorrect operation of the system. Take it seriously, for indeed that is approximately the number of machine language directives in your PC-1500's BASIC ROM!

On the other hand, note that everything there is to know about the operation of this instruction can be learned and appropriately controlled. Thus, if every aspect is carefully considered and applied, then there is virtually nothing that can go wrong (unless the machine itself should fail, which, short of catastrophic device failure, is an extremely rare occurrence). Execution of the ADDA XL instruction will invariably produce the same result as long as all the parameters remain the same each time the operation is performed.

One of the great marvels of the computer is that once a process has been properly modeled (programmed), no matter how many months or years of development behind it, its future use is timeless. It may have taken five people-years of work to create the coding contained in the BASIC ROM in the PC-1500. However, that coding can now be duplicated in literally millions of devices and used by humankind for eternity (if the manufacturer obliges).

Let us not stray too far from our course. The subject at hand is learning the detailed aspects of invoking the ADDA XL directive.

Another point worthy of note: Take heed of the fact that the carry flag acts as an addition to the least significant bit of the registers at the start of the operation. However, at the end of the procedure it is acting as the overflow handler from the most significant bit position of the accumulator! Thus, in this instance, the carry flag appears to serve opposite ends of a register depending on when one "looks" at the situation. At the beginning of the operation, it is serving as an addend at the rightmost bit position. At the conclusion, it is acting as an extension at the left end of the accumulator -- effectively serving as a ninth bit position.

Finally, it will be mentioned that the ADDA XL instruction takes but one byte of memory to fully specify the directive. Other types of instructions, even variations of the "add to the accumulator" instruction such as this one, can take several bytes of memory as will be outlined shortly.

Learn One, Learn a Bunch

A nice thing about studying the machine codes for a particular CPU is that once you have learned the basic operation of one instruction, you have often learned the key facets of a whole group of instructions. That is certainly the case with the ADDA XL instruction. You see there is a whole group of "add to the accumulator with carry"

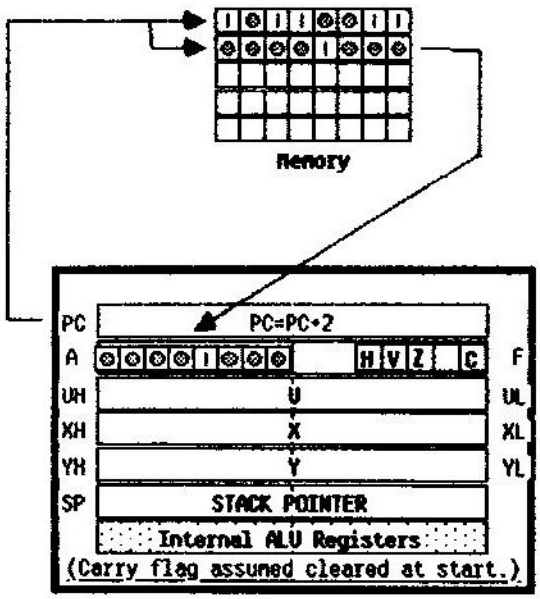
commands:

Mnemonic	Code
ADDA #nn	B3
ADDA (U)	23
ADDA (X)	03
ADDA (Y)	13
ADDA nnnn	A3
ADDA UH	A2
ADDA UL	22
ADDA XH	82
ADDA XL	02
ADDA YH	92
ADDA YL	12

Right there you have 11 different types of ADDA instructions. Can you figure out what each mnemonic represents?

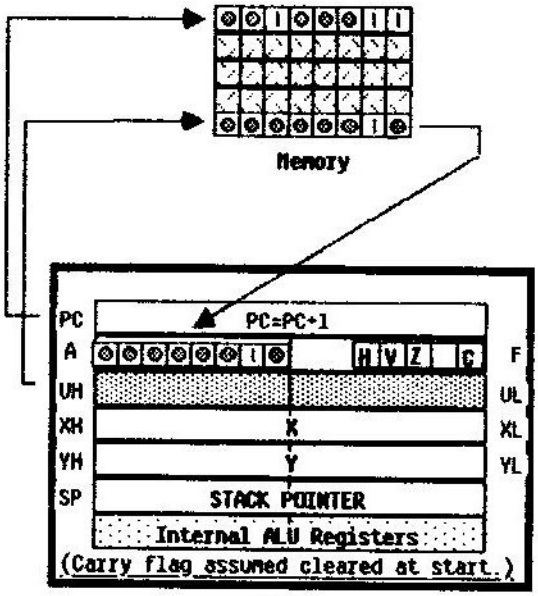
Well, some conventions are being followed with the mnemonics. First of all, the register that follows the ADDA part of the mnemonic is always being added to the A register with the result being left in the accumulator. The register being worked with is left unchanged by the addition operation.

The symbol "#" associated with a mnemonic is a notation that signifies an "immediate" piece of information. An *immediate* piece of data is one that immediately follows the opcode byte. Thus the mnemonic ADDA #nn indicates that whatever value is in the byte that immediately follows the opcode (B3 in this case) will be added to the accumulator. If the hexadecimal value 08 followed the opcode, then 08 would be added to the current contents of the accumulator. Note that this particular instruction thus requires two bytes of memory in order to be fully specified. The code B3, indicating the type of instruction to be performed, must be immediately followed by a second byte containing a value that will be used as an operand. In this case the operand will be added to the contents of the accumulator.



CPU

An ADDA mnemonic followed by a 16-bit register designation enclosed in parenthesis, such as ADDA (U) means that the specified register is to be used as a *data pointer*. That means that the address contained in the 16-bit register will be the address in memory from which the operand will be taken. In other words, register U (in this case) *points* to the memory address where the actual data byte is located. Note that there is an ADDA instruction for use with each of the three (U, X and Y) 16-bit data pointer registers. Also note that the instruction itself only requires one byte of memory. The operand that is pointed to by the appropriate data pointer register can be any location in memory. In many applications a separate area will be reserved or established in memory to serve as a data storage area. The data pointer will then be set up (prior to execution of the ADDA command) to identify the value that is to be used in the addition operation.



CPU

Can you deduce the format of the directive signified by the mnemonic ADDA nnnn? Sure! It means that the opcode is followed by a 4-digit (hexadecimal) address. The contents of that memory address will contain the value that is to be added to the accumulator. Note that now the instruction uses three bytes of memory in order to be completely specified. One byte for the opcode and two to identify the memory address that contains the data byte.

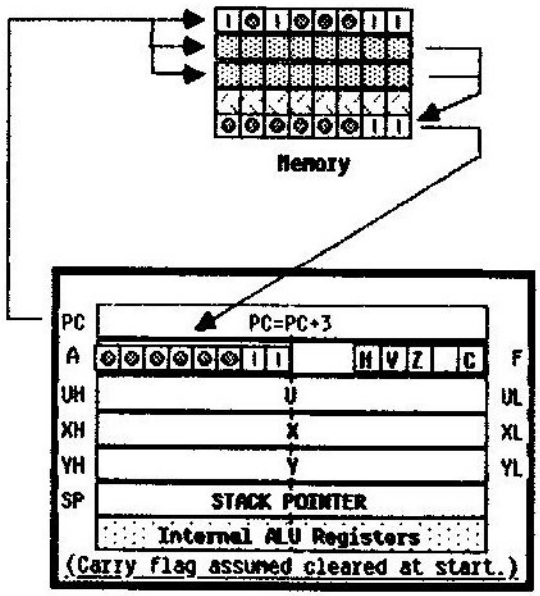
Perhaps this is a good place to delve into thoroughly understanding how many bytes of memory a particular type of instruction is said to

 Copyright 1983 POCKET COMPUTER NEWSLETTER

require. The key here is to realize that the program counter's job is to identify where the next opcode to be executed is located in memory. If an instruction such as ADDA (U) is executed, then the program counter (PC) is merely incremented once to point to the next consecutive memory location. This is because the ADDA (U) directive only uses one byte of memory in which the actual opcode itself is stored.

However, when an instruction such as ADDA #nn is processed, the PC must be incremented twice. Once to advance over the opcode and once to go beyond the immediate byte of data that must be provided after the opcode!

And, the instruction ADDA nnnn will cause the PC to be incremented three times. Once for the opcode, twice for the two bytes following it that hold the 4-digit hexadecimal address where the data value is located.



CPU

In summary, the program counter must automatically be advanced so that it identifies the start of the next instruction that is to be executed. The number of bytes that it must be advanced over for each type of instruction is what we are referring to when we say a directive is a one-, two- or three-byte directive. (While most instructions executed by the LH5801 only require 1 to 3 bytes, there are a few that need even more. Those will be introduced in due course.)

Finally there is a group of ADDA instructions of the type with which you are thoroughly familiar: the group that adds the contents of an 8-bit CPU register (UH, UL, XH, XL, YH or YL) to the accumulator. Note that this latter group consists of data operations that are "intra-CPU." That is, the addition operation takes place amongst registers within the CPU itself. The other types of ADDA instructions discussed all took a data byte from a location in memory.

An Ordered Structure

While this text will not dwell on "hardware" aspects of CPU operation, from time-to-time I may point out features that are related to the internal circuit design. This will be done for the purpose of providing interesting and useful material that may be of value to some machine language programmers.

For instance, let's spend a few moments examining the machine codes that are used to represent the ADDA commands. The codes are shown in hexadecimal notation. The two hexadecimal digits are easily expanded into two adjacent groups of four binary bits. Thus, the hexadecimal code for the ADDA #nn instruction, B3, stands for the binary pattern: 1 0 1 1 0 0 1 1. It is interesting to note that all the ADDA directives associated with intra-CPU register additions end with the hexadecimal digit 2 while all of those that involve data bytes stored in memory end with the digit 3. You can also observe that the most significant hexadecimal digit similarly exhibits interesting structures. For instance, the most significant digit for ADDA (X) is 0, while for ADDA (Y) it is 1 and for ADDA (U) it is 2. Look, too, at how the most significant digit has a distinct pattern amongst the ADDA XH and XL, YH and YL and UH and UL codes. When expanded to a binary representation it is clear that the most significant bit in the opcode is being used to select the high order register of each register-pair.

This information might seem like trivia to beginning programmers, but it can have practical application as you gain coding experience. If you have ever had occassion to watch a seasoned machine language programmer at work (one with several years of concentrated practice) you may have been amazed to notice that the person apparently knew how to directly translate machine code from, say, a raw hexadecimally-coded memory dump. One important way an experienced programmer develops this skill is by learning the significance of particular bit-patterns and the use of individual bits within those patterns.

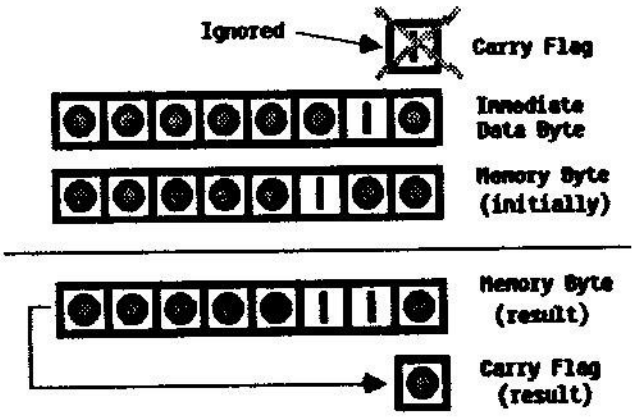
For instance, there are only two other types of PC-1500 ML instructions that end with the hexadecimal digit 2. One of these other types always has the most significant digit in the hexadecimal range C through F. The other uses 4, 5 or 6 as the most significant digit. Knowing this (by heart!) you can instantly classify any machine code ending with a 2! Would you be surprised to know that the group that has 4, 5 and 6 in the most significant digit positions involves the X, Y and U registers in that order? Not if you noticed that the

ADDA group had the most significant digit advancing from 0 to 2 in the same order for registers X, Y and U!

I am not going to recommend that a beginner proceed to put much effort into memorizing this type of information. I am, however, pointing out how it can be done so that you will be aware that the CPU is a logical device, it was designed by a human designer who developed logical patterns to control its operation, and there is nothing magic about the machine. As you gain familiarity with a particular CPU you can start to casually look for such recurring patterns and use the information for what it is worth. If you become a professional machine language programmer, it can be worth a lot of time. However, it is not a necessary skill. You can always look up the meaning of any machine code in a suitable machine code table or better yet, use a computer program known as a disassembler. More about that later.

Add without the Carry

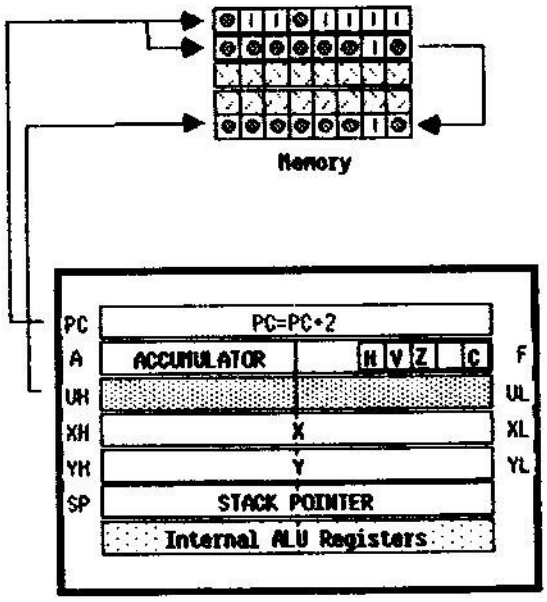
As you might imagine, there are times when a programmer may want to perform addition operations without having to be concerned with the status of the C flag at the start of the procedure. (In other words, without having to be bothered with seeing that the carry flag has been cleared.) The LH5801 CPU has a group of instructions that provides this capability, but it is not as comprehensive as the previously described ADDA category. In fact, it is restricted to commands that add an immediate data value to the contents of a location in memory, instead of the accumulator.



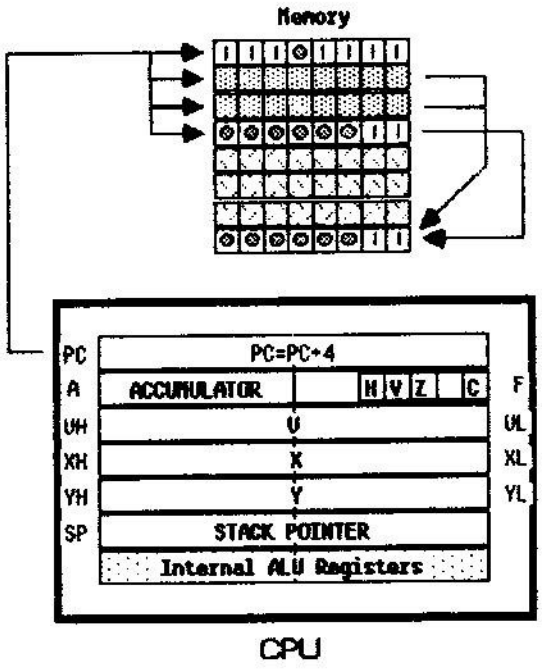
The set has the following mnemonics and machine codes:

Mnemonic	Code
ADNC (U) #nn	6F
ADNC (X) #nn	4F
ADNC (Y) #nn	5F
ADNC nnnn #nn	EF

There are a number of points you must know in order to properly utilize these directives: (1) The addition is performed using two's complement notation, (2) The initial contents of the carry flag are *ignored* at the beginning of the addition process, (3) The result of the addition operation is left in the memory location referenced by the instruction, (4) The status of the carry flag will be determined by the result of the addition operation. It is set if there is an overflow, cleared otherwise. (5) The status of the Z, V and H flags may also be affected by the results of the procedure.



The first three instructions in this group are two-byte directives. One byte representing the opcode, the second the value that is to be added to the specified location in memory. Note that the 16-bit registers U, X or Y must contain the address of the byte in memory to which the immediate data is to be added.



CPU

 Copyright 1983 POCKET COMPUTER NEWSLETTER

The last directive, ADNC nnnn #nn is an example of a directive that will cause the program counter to be advanced by a count of *four* when it is executed. Once for the opcode, twice for the 16-bit memory address in the next two bytes, and a fourth time for the immediate data byte!

Note the machine code pattern. The lower nibble is F for the entire class of instructions. The higher nibble goes from 4 to 5 to 6 to signify use of the X, Y and U registers respectively.

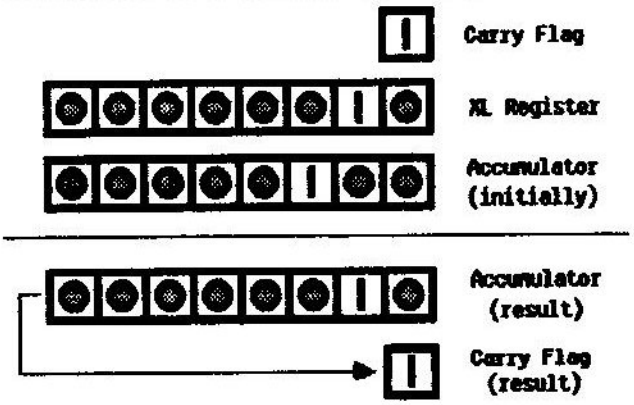
This type of instruction can be particularly valuable when it is desired to advance an address pointer that is being stored in memory. This is especially true when one is in a situation where it might be beneficial not to have to alter the contents of the accumulator.

Subtraction

You will undoubtably not be surprised to learn that the LH5801 CPU is able to perform subtraction. In fact, it has an entire group of directives similar in structure to that of the "add with carry" group that has already been discussed. Here are the mnemonics and machine codes used to represent these subtraction directives:

Mnemonic	Code
SUBA #nn	B1
SUBA (U)	21
SUBA (X)	01
SUBA (Y)	11
SUBA nnnn	A1
SUBA UH	A0
SUBA UL	20
SUBA XH	80
SUBA XL	00
SUBA YH	90
SUBA YL	10

All of the subtraction directives take place between the accumulator and a specified register or memory location. Here are the important things to know about these directives: (1) The arithmetic convention utilized is known as two's complement notation. (2) The complement of the carry flag is considered as a *borrow*. (3) The contents of the specified register or memory location are always subtracted from the contents of the accumulator. (4) The result is left in the accumulator. (5) The original register or memory location value is left unchanged. (6) Any "underflow" caused by the subtraction process will be reflected in the status of the carry flag at the end of the operation. (7) The status of the Z, V and H flags may be altered by the operation.

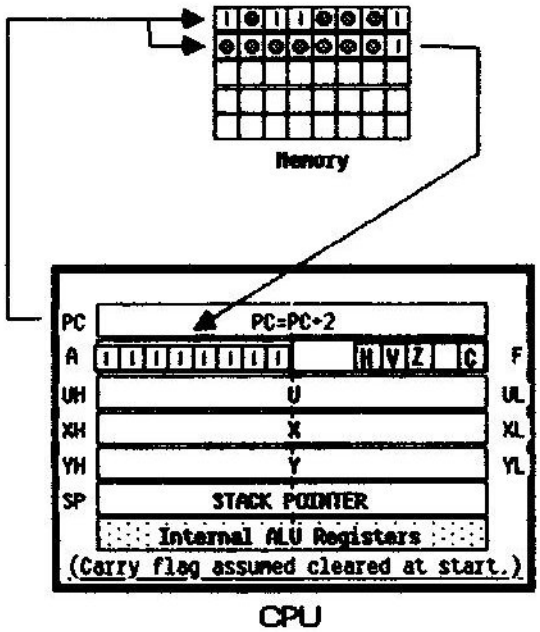


Addressing

As was the case with the addition directives, the subtraction instructions take one, two or three bytes depending on the *addressing* involved. The group of subtraction instructions listed represent four different types of addressing modes.

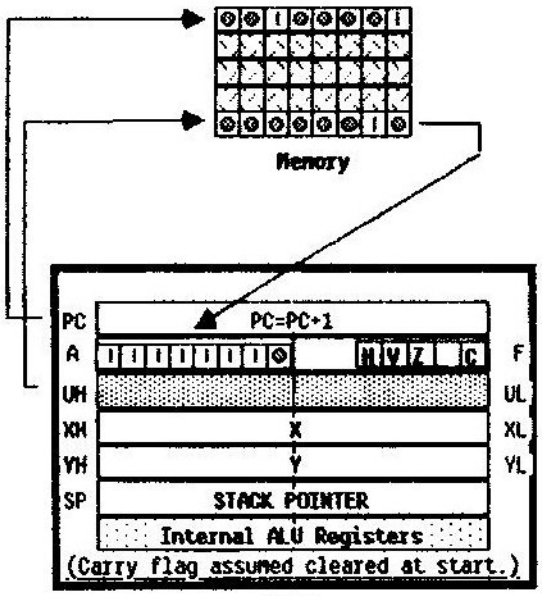
Those such as SUBA UH and SUBA UL utilize what is known as *implied* addressing. That is, the opcode itself specifies the registers or locations that are affected by the operation. The hexadecimal opcode 20 (which is 0 0 1 0 0 0 0 0 in binary) tells the CPU everything it has to know in order to perform the directive. When it detects that pattern, it "knows" that it is to perform a subtraction operation using the accumulator and the 8-bit UL register. In a like fashion, the pattern 1 0 1 0 0 0 0 0 (A0 hexadecimal) tells the CPU to perform the subtraction between the 8-bit UH register and the accumulator. These types of directives only use one byte of memory since the addressing information (related to what registers will be affected) are inherent within the actual machine code. In fact, this mode of addressing is sometimes also referred to as *inherent* or *register* addressing.

The directive SUBA #nn is an example of *immediate* addressing. The byte that immediately



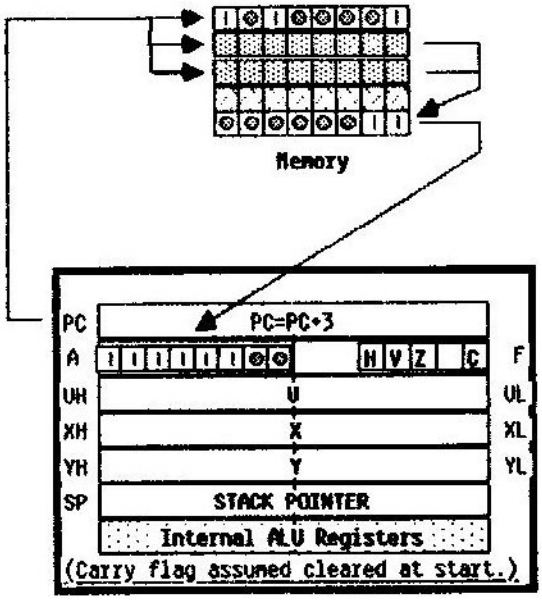
follows the opcode is the location that is to be subtracted from the accumulator. This directive requires two bytes of memory. One for the opcode, the other to hold the data that is to be subtracted.

The instructions SUBA (U), SUBA (X), SUBA (Y) all represent examples of what is often referred to as *register indirect* addressing. This is because the contents of the 16-bit register U, X or Y holds an address of a memory location where the actual data that is to be subtracted is stored. Note that the U, X or Y registers themselves do not hold the data, rather they *point* to the data location.



CPU

Finally, SUBA nnnn is illustrative of the mode known as *direct* addressing. In this mode, the two bytes that follow the opcode contain the actual address indicating where the data byte is stored.



CPU

Note that these two address bytes are considered as part of the instruction. Thus, this directive takes three bytes of memory to be properly defined.

These *addressing modes* (and there will be a few others introduced later) are an important concept. Most types of instructions have several different possible addressing modes. If you go back now and review the ADDA group of instructions you can observe that it has the exact same set of addressing modes.

Some instructions utilize a combination of addressing modes. The previously discussed ADNC (U) #nn utilizes a *register indirect - immediate* mode. The 16-bit register (U in this case) points to a memory location and the immediate data byte holds the data that is to be added to the contents of the referenced memory location. Can you figure out what combination of modes are used in the ADNC nnnn #nn instruction? Right! It combines *direct* and *immediate* modes.

Can you find any connection between the machine codes used (in the opcode byte) and the addressing mode being invoked? Remember, the CPU was created by a logic-oriented circuit designer. Do you think that perhaps a particular bit position within the opcode byte is frequently used to signify a particular addressing mode? You might keep an eye out for address-mode patterns (in the opcodes) as additional types of instructions are introduced. This is another piece of information that the "pro's" look for as they become intimately familiar with the operation of a particular CPU.

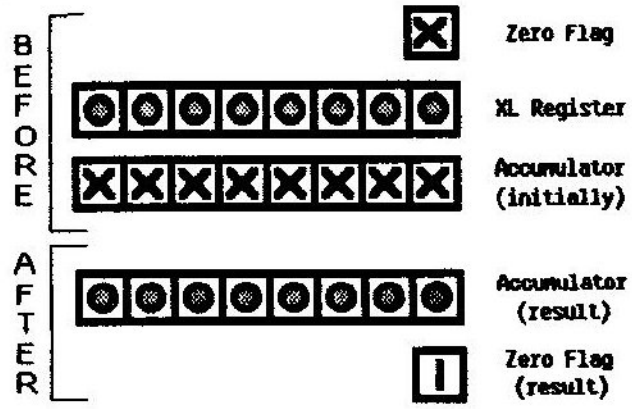
Loading the Accumulator

Adding and subtracting quantities in a computer is all well and good, but you need to have values to work with in the first place. How do you obtain values and get them positioned in the proper registers? You use *load* and *store* directives. These are instructions that cause information to be moved about from one location to another. They are probably amongst the most often used commands. Right now we will become familiar with the group of directives that may be used to place information into the accumulator or A register:

Mnemonic	Code
LDA #nn	B5
LDA (U)	25
LDA (X)	D5
LDA (Y)	15
LDA nnnn	A5
LDA UH	A4
LDA UL	24
LDA XH	84
LDA XL	D4
LDA YH	94
LDA YL	14

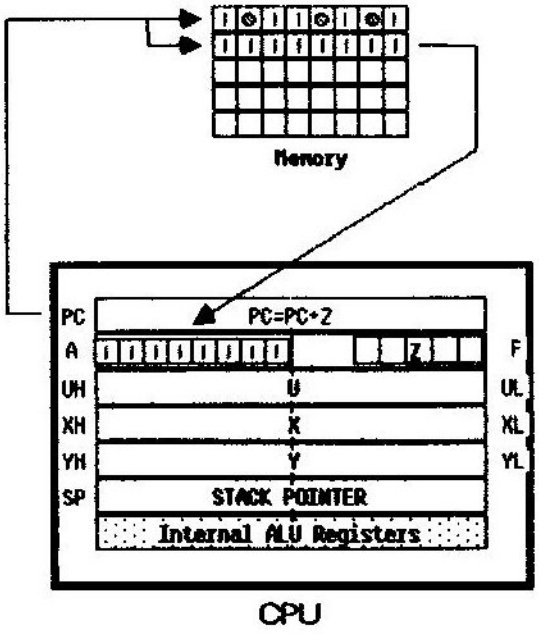
 Copyright 1983 POCKET COMPUTER NEWSLETTER

Here is what you need to know about this group of instructions: (1) A "copy" of the information in the specified memory location or register is always transferred (loaded) into the accumulator. (2) The status of the Z (zero) flag will reflect whether or not the value transferred was equal to zero. (3) The other flags are not affected by this type of operation.



X = Does not matter!

There, that was easy wasn't it? All these instructions do is provide a means of moving information about in the computer system. Note that you can transfer data from the CPU registers and (using several addressing modes) from locations in memory. This particular group always places the data into the accumulator.



You might also want to take note of the fact that only one CPU flag is affected by this type of directive: the zero (Z) flag.

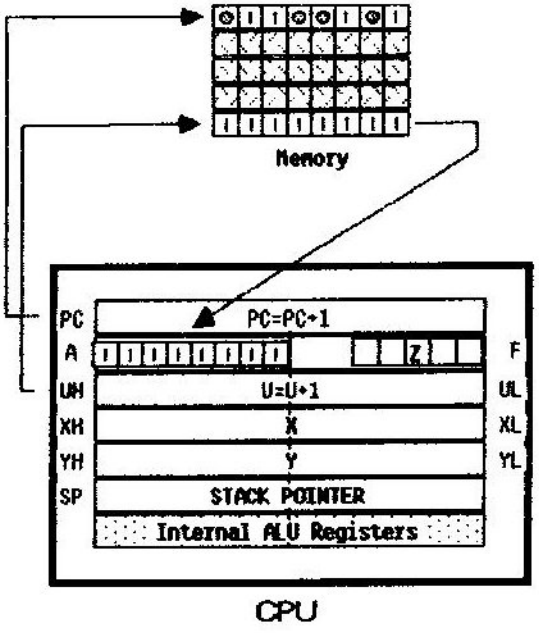
Can you think of any reasons why the designers allowed only this flag to be affected by the load operations?

There are several good reasons. One is that some of the other flags, such as the carry, need to be maintained when multiple-precision arithmetic is being performed. (That is arithmetic where the numbers being manipulated are stored in two or more bytes, yet are considered as one value.) Because these numbers take more than one byte in which to be stored, they have to be processed in byte-sized sections. If a load directive interfered with the status of, say, the carry flag, then the add with carry instructions would be worthless during such operations.

But why let even the zero flag be influenced by load operations? Because, as will be illustrated later, being able to detect a "zero byte" (a byte in which all the bits are cleared to zero) has many practical applications. It is often used to denote the end of data in lookup tables, to mark the end of a string of stored text, and so forth. So, being able to easily detect the presence of a zero byte when it is first loaded into a location, can have considerable practical programming value. The LH5801 designers knew this and thus permitted the zero flag to be affected by this operation, while protecting all other flags.

The load directives are used so extensively in machine language programming that it is common to provide supplemental types of instructions. For instance, the CPU used in the Sharp PC-1500 has the following special load directives:

Mnemonic	Code
LDAI (U)	65
LDAI (X)	45
LDAI (Y)	55
LDAD (U)	67
LDAD (X)	47
LDAD (Y)	57



These are known as *load and increment* and *load and decrement* directives. They transfer a byte from the memory location pointed to by the contents of the 16-bit registers (U, X or Y) into the accumulator *and then they automatically increment or decrement the pointer register by a count of one!* These powerful instructions are most useful when it is desirable to sequentially process a whole block of memory.

Only the Z (zero) flag can be affected by these directives, in exactly the same manner as for the LDA (U) instruction. That is, it is cleared if the byte transferred is non-zero and set if it is zero. Notice that the increment or decrement of the pointer register does not affect the flags. *It is the byte being transferred* that controls the status of the Z flag when any of this group is executed.

All of the load instructions discussed so far have been designed to transfer data *into* the accumulator. But, how about getting data into the other CPU registers?

Loading Other CPU Registers

Each half of the 16-bit registers U, X and Y can be loaded with immediate data by the following load instructions:

Mnemonic	Code
LDUH #nn	68
LDUL #nn	6A
LDXH #nn	48
LDXL #nn	4A
LDYH #nn	58
LDYL #nn	5A

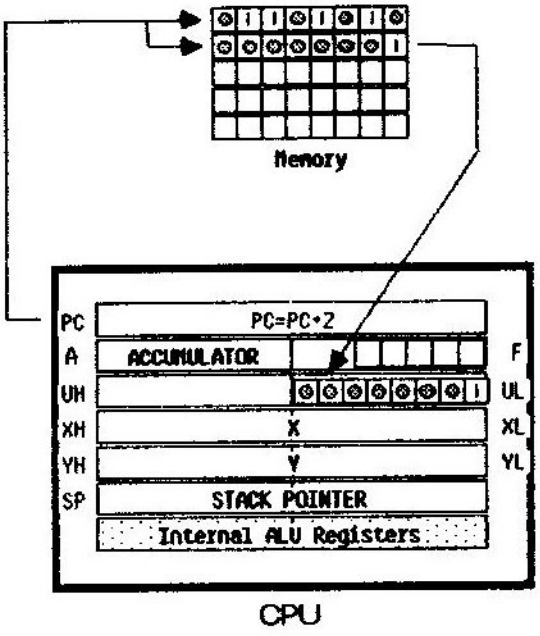
One important difference between this group of directives and the previously described load commands that dealt with the accumulator is that *none of the flags* are altered by these transfers, not even the zero flag.

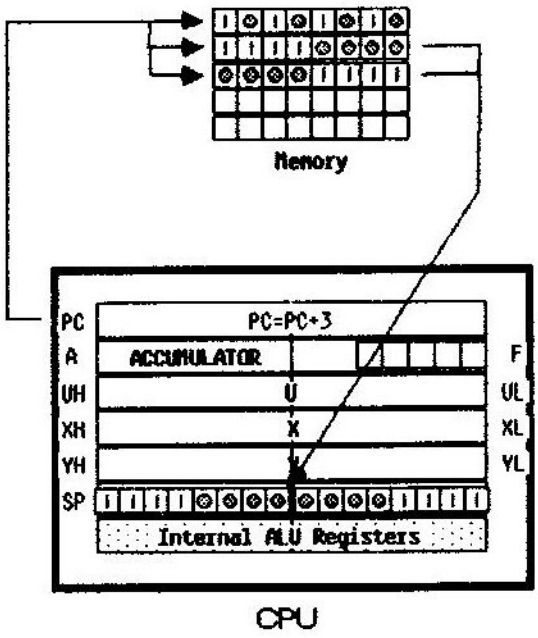
The fact that each half of the 16-bit pointer registers can be loaded with a byte has many practical applications. Frequently these registers are used as temporary storage locations within the CPU. Being able to treat these registers as consisting of two independent bytes (when it is desired by the programmer) means that more data can be readily accessed within the CPU.

On the other hand, you must remember that if you want to set up a 16-bit register so that it contains a memory address, then you must use a sequence of two load immediate instructions. One to load the most significant half (UH, XH or YH) and the other to load the least significant half (UL, XL or YL).

It would have been theoretically possible for the CPU designer to have provided a load immediate instruction that loaded all 16 bits of the data pointer registers at one time. The advantage of opting for the 8-bit loads is flexibility. The trade-off is that it takes two loads to completely define an address. (However, as you will learn later, this is not as serious a trade-off as it might initially appear. In practical programming situations it turns out that the most significant portion of an address does not have to be changed as often as the least significant portion. Thus, many times, the address in a data pointer register can be altered to the desired value just by changing one byte.)

There is, in fact, a 16-bit load immediate directive provided in the instruction set of the CPU being discussed. It is used to load the stack pointer with a complete 16-bit address.





 Copyright 1983 POCKET COMPUTER NEWSLETTER

Mnemonic	Code
LDS# nnnn	AA

This is the *only* directive in the instruction set where the two bytes following the opcode both act as immediate data values. Normally these values represent an address that is loaded directly into the stack pointer. None of the CPU flags are affected by this command. (The stack pointer will be discussed in detail at a more appropriate point. Have patience, please.)

Storing the Accumulator

So far, all the "load" instructions have been transferring information in one direction: into the accumulator or from a location in memory to a register in the CPU. It would be a pretty useless computer if data could only be transferred into the accumulator or CPU and not out. Of course, that is not the case. There is another group of instructions that can transfer information in the other direction. In some systems, this group is still referred to as "load" directives. Here, however, to differentiate between the directions of data flow, an instruction that passes data *from* the accumulator to a location in memory, will be designated a *store* directive.

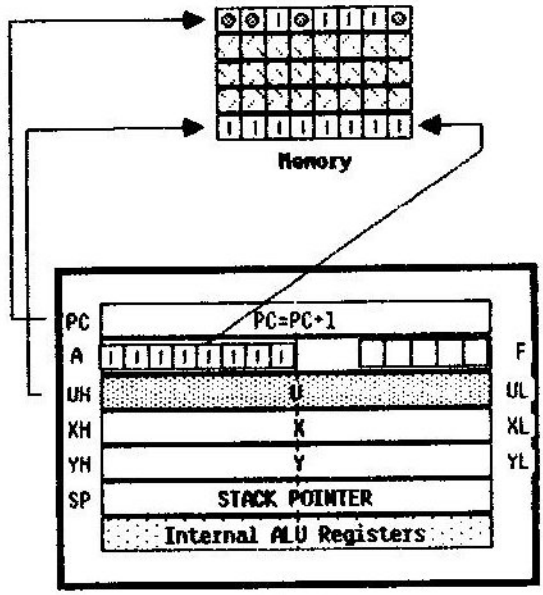
The first of the store directives to be discussed are those that simply transfer the contents of the accumulator to a location in memory or to an 8-bit register within the CPU:

Mnemonic	Code
STA (U)	2E
STA (X)	0E
STA (Y)	1E
STA nnnn	AE
STA UH	28
STA UL	2A
STA XH	08
STA XL	0A
STA YH	18
STA YL	1A

You need to know the following in order to use these commands effectively: (1) A copy of the A register (accumulator) is placed into the indicated memory location or 8-bit CPU register. (2) *None* of the flags are affected by these operations. Not even the zero (Z) flag.

Thus, the CPU can store results from the accumulator into memory or another CPU register without altering the status of any of its flags. This ability to accomplish such transfers without altering the flags is useful in many circumstances. Note that now not even the Z flag is sacrificed. This is because when *constructing* tables or text strings, the CPU can insert a zero byte to serve as a terminator. It does not have to *detect* such a marker as is typically the case when memory is being scanned. The distinction will be made clearer later.

Remember, the STA (U), (X) and (Y) directives use the contents of those 16-bit registers to *point* to the location in memory where the byte will be saved.



CPU

The STA nnnn command uses the 16-bit value (nnnn) as an absolute memory address in which to store the data.

The store directives also include the so-called *auto-increment* and *auto-decrement* groups:

Mnemonic	Code
STAI (U)	61
STAI (X)	41
STAI (Y)	51
STAD (U)	63
STAD (X)	43
STAD (Y)	53

These are similar to the load directives. After a byte is transferred from the accumulator to the memory location pointed to by the address in the U, X or Y register, the contents of the data pointer register is automatically incremented (by one) or it is decremented (by one). It is thus ready to point to the next sequential location in memory. Naturally, this is quite beneficial when it is desired to pack data into a continuous block of memory. However, unlike the load instructions, the CPU flags are not altered by these store commands.

In and Out

Finally, to wrap up the presentation of the load and store instructions, there is a real whiz-bang directive. This command does the equivalent of

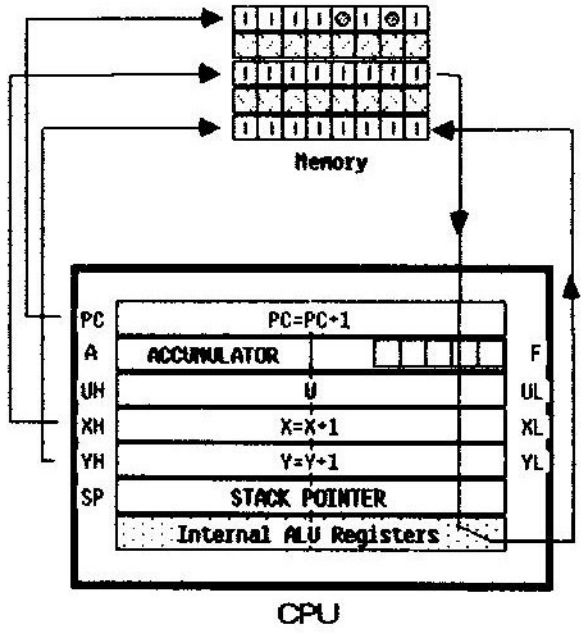
both a load and a store plus it automatically increments *two* data pointer registers! It is a real block-buster -- and that is exactly what it is intended to do -- move large blocks of memory quickly and conveniently.

Mnemonic	Code
STI (X)(Y)	F5

The byte of data stored at the memory location pointed to by the contents of 16-bit CPU register X is first loaded into an *internal* CPU register. It is then stored into the memory location pointed to by the contents of the 16-bit CPU register Y. Next, the contents of both X and Y are advanced by a count of one.

None of the CPU flags are affected by the operation of this instruction.

It is a super instruction to use when you need to move blocks of data from one location in memory to another! This type of directive is indicative of advanced microprocessor designs. Earlier CPU chips did not have such powerful operations combined into a single machine language opcode.



CPU

A Program

Only a small fraction of the commands that are available in the repertoire of the LH5801 have been presented at this point. There is much more to learn. But perhaps now is a good time to get a glimpse at what the future looks like for a machine language programmer. More than enough instructions have been explained so that we can demonstrate a rudimentary, but actual, program. The purpose at this point is to build confidence and provide some feedback for the serious student. Take heart, what you are learning really works. You can make things happen with this information!

The goal at this point is to develop a series of instructions that will accomplish the following: Add one number to another and store the result in a location in memory.

Simple enough, eh? How many ways do you think that could be accomplished using just the machine language directives you have learned so far? There are quite a few, believe me. In fact, that is one of the joys of working at the machine level. There are usually a variety of ways of accomplishing any given task. The "best" way often depends on your objectives. Sometimes, there is really no such thing as a best way. There are, instead, simply a host of ways to accomplish the same end result.

Before reading further, you might just want to take a few minutes to write down some of the ways that you can think of to accomplish the goal: add two simple numbers together and store them in memory.

Did you think of the following method?

(1) Load one value into the A register.

(2) Store the contents of the A register (which now contains the first value) into a specific memory location.

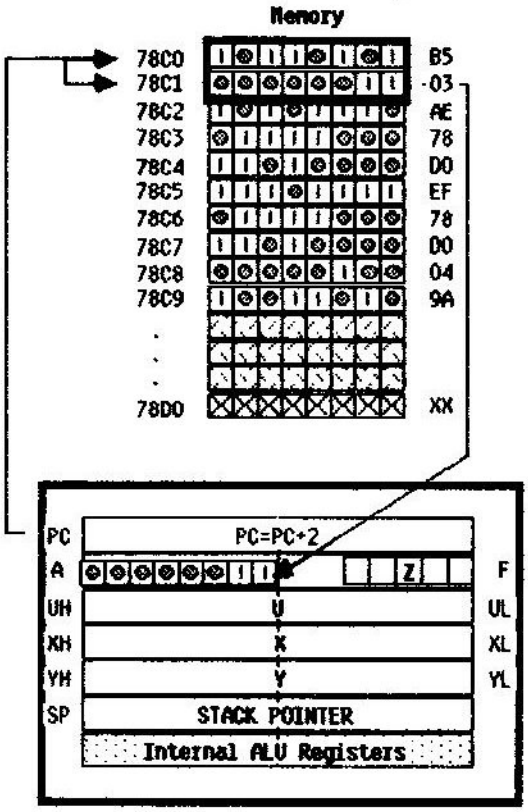
(3) Add the second value to that memory location, leaving the result there.

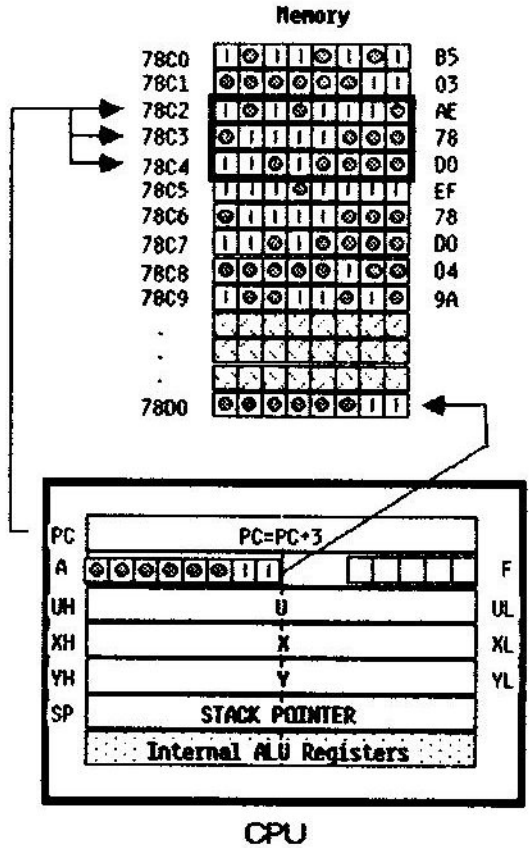
One reason for choosing this method is that it is not affected by the status of the carry flag at the start of the operation. Do you recall that all of the instructions that add something to the *accumulator* are dependent upon the initial status of the carry flag? How do you know, at any given point, what the status of the carry flag is (or will be)? If you don't know what it is, should you be using an instruction that is dependent upon its status? (Actually, there are instructions that can be used to place the carry flag in a known state. And, it is often possible to logically control the state of the flag at a particular point in a program. However, if you don't know what those methods are, then you had better be careful when dealing with "add with carry" commands!) Did you choose a safe way of accomplishing the given task or would you have been on unsure grounds? *You were on unsure grounds if you elected to use any add with carry directive and you did not specifically devise a way of insuring that the carry flag was set to a known state prior to performing the addition!*

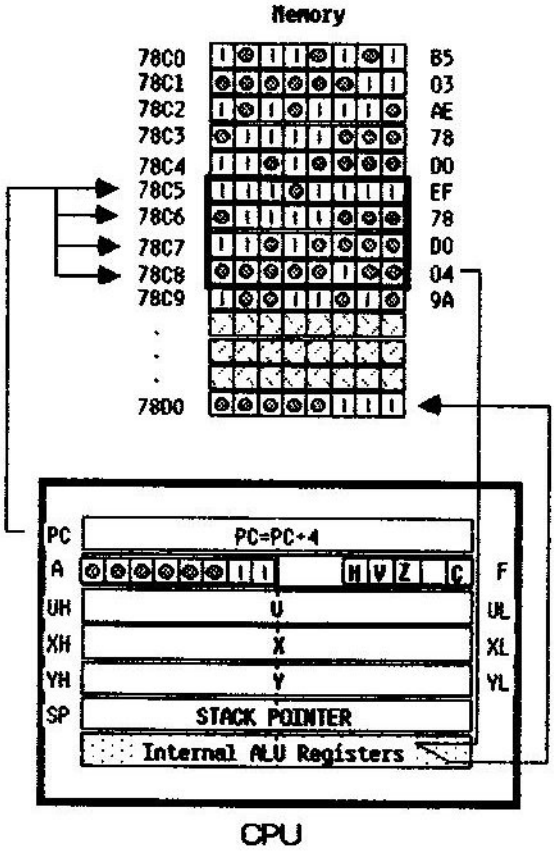
The first rule of machine language programming is that you have to *know* what is going on. Guessing or hoping that conditions will be this way or that way will not work. If you can't be sure about the status of something, then you had better design things so that you end up *knowing* the status. If you cannot logically figure out a way of accomplishing this, then the machine cannot do it either. (You see, *you* really *are* the boss.)

 Copyright 1983 POCKET COMPUTER NEWSLETTER

Three directives would accomplish the stated goal: LDA #nn, STA nnnn, ADNC nnnn #nn. Load the accumulator with the first number. Store the accumulator in the desired memory location. Add







(no carry) the second number to the memory location where the first number was stored.

If the numbers to be added were 3 and 4 and the hexadecimal address of the storage location in memory was 78D0, here is how the machine code for this series of instructions would appear (in hexadecimal notation): B5 03 AE 78 D0 EF 78 D0 04.

The accompanying pictorials illustrate each step of the program's operation.

You can try this program out for yourself by loading it into a PC-1500 or PC-2. One way to do this is to use BASIC POKE statements. Here is a BASIC program that will load and then execute those machine language directives:

```
10:POKE &78C0,&B5
   ,&03,&AE,&78,&
   D0,&EF,&78,&D0
   ,&04,&9A
20:CALL &78C0
30:PRINT PEEK &78
   D0
40:END
```

Line 10 of this program places the actual machine codes (using hexadecimal notation) into the PC's memory. These instructions are being stored in the portion of memory normally used to hold the BASIC variable A$, starting at the hexadecimal address &78C0. The values following the four-digit address are the actual machine

codes for the program in hexadecimal format. (The last byte in line 10, &9A, is the code for a machine language directive that will cause the machine language program to be exited back to BASIC.)

Line 20 causes the machine language codes to be executed starting at hexadecimal address 78C0 (where the program was stored). After executing those machine code directives, control returns to the BASIC program.

Line 30 causes the PC to display the contents of the memory location having the hexadecimal address 78D0. This is the first byte of the area in which the BASIC variable B$ is normally stored. The machine language program uses this location to store the first number (from the accumulator). It is also the location where the ADNC instruction takes action and leaves the results of the addition (without carry) operation.

Line 40 denotes the end of the BASIC code.

Executing this BASIC program by placing the PC in the RUN mode and executing GOTO 10 or RUN should result in the value 7 being displayed.

Do you understand everything that is going on at this point? Does everything in the descriptive pictorials that portrays the operation of this machine language procedure make sense? If not, now is a good time to review the earlier parts of this text.

By the way, attempting to examine the operation of machine language routines by using PEEK and POKE directives is a very slow and tedious method. A far better tool to use is a machine language monitor program. This is a special program designed to facilitate working at the machine code level. If you plan to make a serious study of this discipline, I strongly recommend that you obtain a copy of the **Loader/Monitor/Disassembler** package sold by the *POCKET COMPUTER NEWSLETTER*. This is a powerful tool that makes it easy to place machine codes into memory, examine the contents of memory, execute machine language routines, and so forth, in a direct mode, without having to translate through PEEK and POKE directives. The disassembler program which is part of this package utilizes the *Rober Mnemonics* which are used in this text. It is able to translate machine codes directly into these mnemonics. Hence, it is highly useful for verifying that machine language routines have been properly loaded into memory as well as for use in exploring "uncharted" areas of memory (such as ROMs). The loader part of this package makes it easy to put together sections of a machine language program. Plus, the package provides a convenient means of saving and restoring machine code programs by using a cassette recorder. No serious MLP student should be without such a tool.

Is learning to program in machine language worth all the trouble? Not for everybody. It depends on what you want to end up being able to do. You now can probably begin to see that dealing at the machine level greatly complicates matters from the programmer's view. It took nine bytes of memory storage just to specify the adding together of two tiny numbers. Even with such a simple task, one had to be concerned with the use of the carry flag. The work is greatly compounded when one has to start dealing with larger values or non-integer quantities.

But, you can probably also gather that the degree of control at this level is fantastic. You are able to dictate every aspect of the machine's operation. Streamlining the flow of operations to accomplish a specific job is just one noticeable benefit. Speed of operation is another. Do you realize that those three instructions used to add together two simple numbers can be performed in about 25 millionths of a second? You could string together some 40,000 similar sequences and still have them all performed in less than a second!

If you still want to hang in there and learn more about the subject, then read on. It is time to describe some of the logical operations that the LH5801 CPU can perform.

The Logical AND

The next few classes of instructions that will be discussed are those known as *Boolean logic* operations. In the LH5801 these mathematical operations are always performed on the contents of the accumulator or on the contents of a location in memory.

The ability to perform these types of logic operations are valuable in many applications. Indeed, they give the computer the ability to duplicate the type of electronic logic found in modern electronic digital circuitry.

These logic operations are always performed on a bit-by-bit basis between the accumulator (or a memory register) and the operand byte. The results of the operation are stored in the accumulator (or memory register). Furthermore, the status of the Z (zero) flag will be affected by the results of the logic operation. Thus, these types of directives, among others, ultimately provide the computer with a means of modifying its own behavior *depending on its findings* as it examines data.

The first group of Boolean logic operations to be presented are those that perform a logical AND operation between the accumulator (A register) and another byte of data in memory.

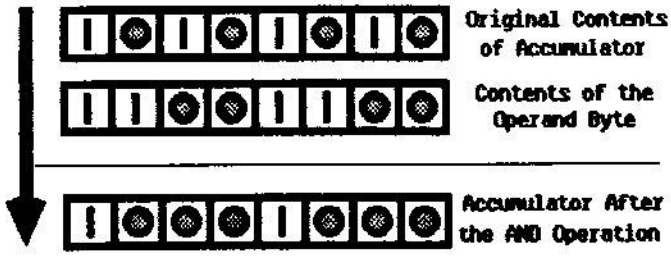
Mnemonic	Code
ANDA #nn	B9
ANDA (U)	29

 Copyright 1983 POCKET COMPUTER NEWSLETTER

ANDA (X)	09
ANDA (Y)	19
ANDA nnnn	A9

Note that there are three types of addressing provided: immediate, register indirect, and direct. To save space in the future, pictorials showing the addressing modes for each class of instruction will not be provided. Refer to the earlier diagrams when necessary to refresh your memory concerning addressing relationships.

The execution of one of the above Boolean AND directives by the LH5801 CPU does the following. Each bit position in the accumulator is compared to the corresponding bit position in the operand byte. As this is done, a logical AND operation is performed between the identically-positioned bits. If both the bit in the accumulator and the bit in the operand are set to the logic 1 state, then the bit in the accumulator will be left in the logic 1 condition. For all other possible combinations (the bits are opposite in state or both are zero), the bit in the accumulator will be left at the logic 0 (cleared) state. If all of the bits in the accumulator end up being zero, then the Z flag will be set. Otherwise it will be cleared. None of the other CPU flags are affected by the operation. The contents of the operand byte are not altered by the procedure.



The logical AND operation may also be performed upon the contents of a particular memory location through the following commands:

Mnemonic	Code
AND (U) #nn	69
AND (X) #nn	49
AND (Y) #nn	59
AND nnnn #nn	E9

The same procedures apply with these directives except that now the operation takes place between the designated memory location and the immediate data byte that is part of the instruction. The results are left in the designated memory site. Only the Z flag is influenced by the operation.

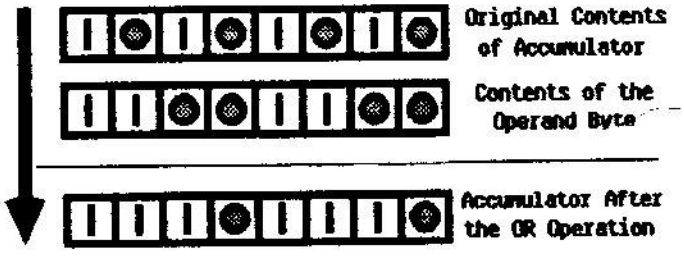
Logical AND operations are particularly useful for performing what are known as *masking* or *stripping* operations. That is, when it is desired to eliminate just a portion of a register's contents. For instance, if one wants to retain just the four least significant bits of an 8-bit register, the entire contents of that register may be ANDed with a register containing the hexadecimal value 0F. Try it yourself (mentally or with the help of pencil and paper) to see how the unwanted half of the byte is reduced to zero. Masking or stripping operations are commonly used to pack data, manipulate binary-coded-decimal (BCD) values and format data inputs/outputs.

The Logical Or

As was the case for the logical AND directives, the logical OR instructions may be divided into two groups: those that leave the result in the accumulator and those that leave the result in a memory location. The first group has the following mnemonics and codes:

Mnemonic	Code
ORA #nn	BB
ORA (U)	2B
ORA (X)	0B
ORA (Y)	1B
ORA nnnn	AB

The Boolean OR operation is also performed on a bit-by-bit basis. For the above instructions, the operation takes place between the accumulator and the designated operand byte with the results being left in the accumulator. A bit position in the accumulator is set to a logic 1 if either it *or* the corresponding bit position in the operand is set to a logic 1. Note that the case where they both are a 1 also satisfies the condition. But, if neither register contains a 1 in a given bit position, then the accumulator bit for that position remains in the logic zero state.



The zero (Z) flag is affected by the results left in the accumulator following the execution of one of these directives.

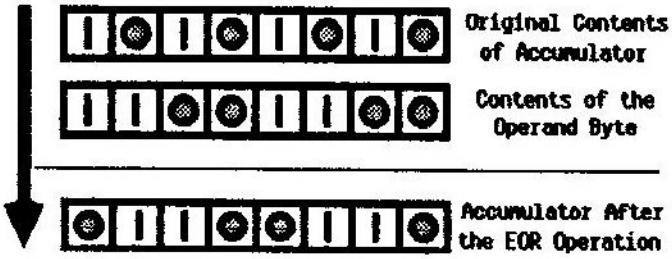
A second group of instructions is performed with and stores the results of the logical OR operation in a designated memory location:

Mnemonic	Code
OR (U) #nn	6B
OR (X) #nn	4B
OR (Y) #nn	5B
OR nnnn #nn	EB

Again, the Z flag is affected by the results of the procedure.

The Logical Exclusive OR

There is a variation of the logical OR operation that the LH5801 is able to perform. It is known as the exclusive OR. It is similar to the OR operation. The difference is that when the corresponding bits between the operand and affected register are both a 1, then the bit position in the results register is cleared to the zero state. Stated another way: a bit in the results register is set to a logic 1 *if, and only if,* just *one* of the registers has a 1 in that bit position.



Because this class of instructions is more limited in application, the CPU designers limited it to operating with the contents of the accumulator and an operand byte. There are no exclusive OR directives that use a memory location as a results register, as there was for the AND and OR classes.

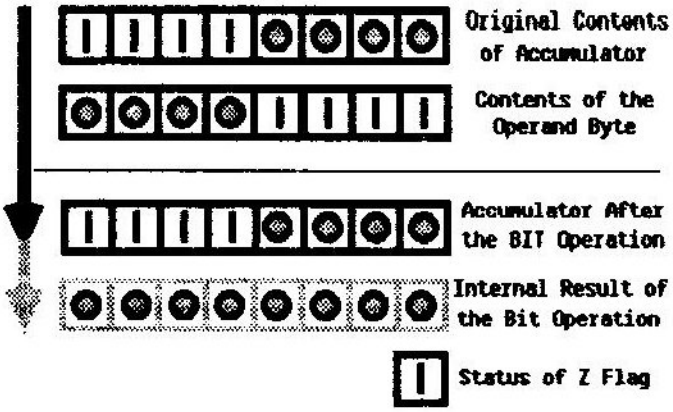
Mnemonic	Code
EORA #nn	BD
EORA (U)	2D
EORA (X)	0D
EORA (Y)	1D
EORA nnnn	AD

The Z flag is the only CPU flag affected by the results of these operations.

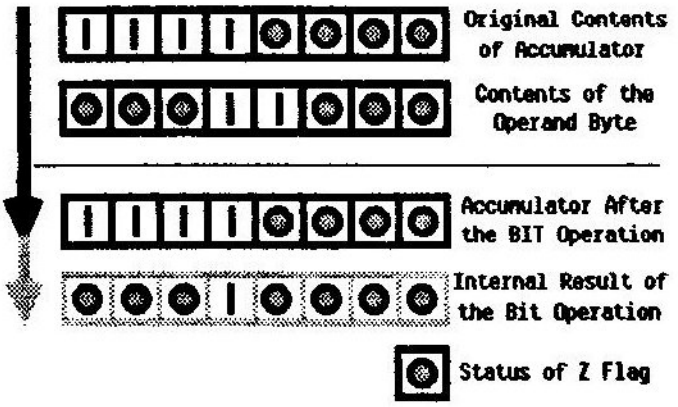
The Logical Bit Test

Another class of instructions provided by the CPU used in the PC-1500 and PC-2 is really just a specialized form of the logical AND operation. The directives in this group enable a programmer to determine which individual bit(s) within a register are set to the logic 1 state. This can be done *without actually altering the contents of the register being examined.*

Known as the *logical bit test,* this type of instruction performs a logical AND between the contents of the accumulator or a location in memory and the contents of another register. The second register can be a location in memory or represented by an immediate data byte, depending on the type of addressing being used. However, the contents of the accumulator or location in memory are not actually altered by the test. Instead, an internal CPU register holds the result of the logical AND operation. If the result held in this internal register is that all the bit positions are zero, then the zero (Z) flag is set.



If any bit position is in the 1 state, then the zero flag is cleared.



This type of instruction is particularly useful in determining if a particular bit within a byte is set. Note, however, that it is the status of the Z flag alone that reflects this information. None of the bits within the accumulator or memory are altered. This operation is an example of what is often called a *non-destructive* test or comparison. This is because none of the information involved in the test is altered or lost.

If you compare the operation of this type of directive against that of the regular logical AND, you will note that the value in the results register (the accumulator or memory location) can be altered by the procedure.

Here are the bit test instructions that the CPU is able to execute in a PC-1500 or PC-2:

Mnemonic	Code
BITA #nn	BF
BITA (U)	2F
BITA (X)	0F
BITA (Y)	1F
BITA nnnn	AF

BIT (U) #nn	6D
BIT (X) #nn	4D
BIT (Y) #nn	5D
BIT nnnn #nn	ED

Compares

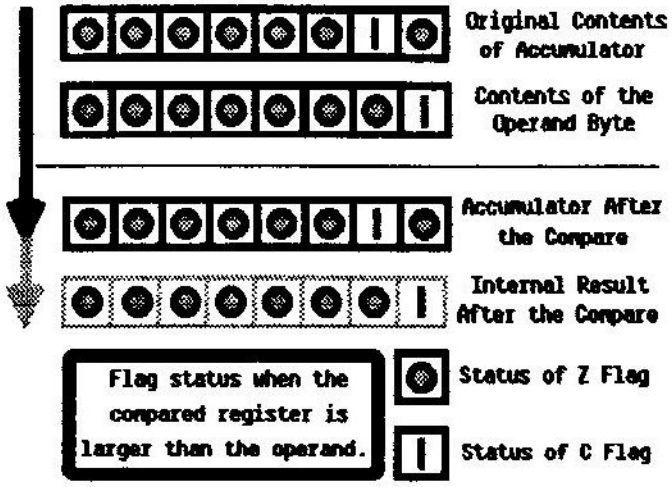
The next series of instructions to be discussed are similar in one concept to the bit test group. That is they also utilize an internal (virtual or invisible) register to store the result of a comparison. The status of the CPU flags are then changed to reflect the conditions in this internal register. None of the regular CPU registers or memory locations that are accessible to the programmer are actually modified by the operation of these instructions.

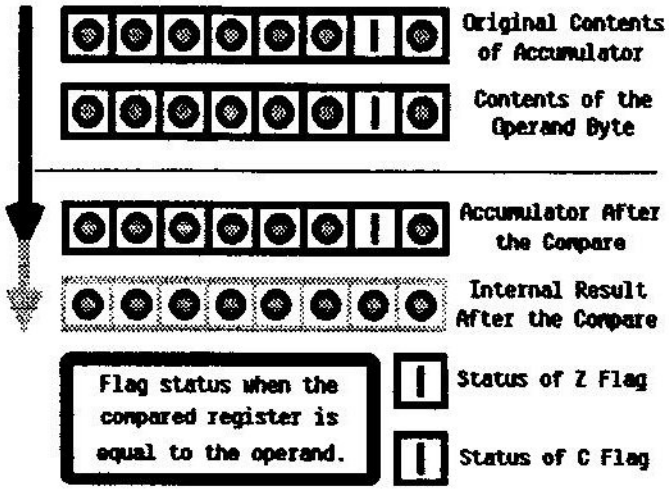
However, these instructions differ in the manner in which the comparison is made. In the bit test group, a logical AND was used to compare bits. In the compare directives, the operand value is *subtracted* (using the two's complement method) from the accumulator, another CPU register or a memory location. The results of this subtraction are held temporarily in an internal (invisible) CPU register. The carry (C) and zero (Z) flags are then set according to those results. (The V and H flags may be altered by these operations, too, but their settings would not be of value after such procedures.)

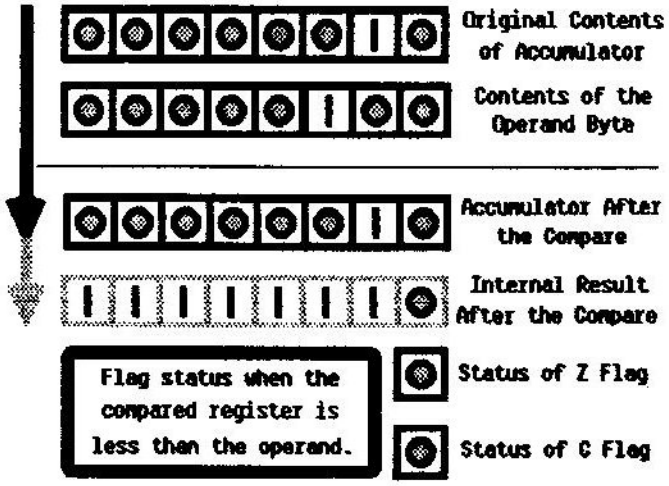
The carry flag is set after the comparison if the result of the comparison indicates that the *operand* was less than the compared value or was equal to the compared value. The carry flag is cleared after the comparison if the operand was greater than the compared value.

The zero flag is set if the comparison indicates that the values were equal. Otherwise it is cleared.

The comparison instructions are among the most important directives in the entire command set in one particular regard: decision making. These instructions enable the computer to make decisions based on the results of comparisons. Other directives, to be introduced shortly, can then







cause the computer to execute a different series of instructions (thus modifying its behavior) based on the decisions made as a result of comparisons.

Since these instructions are so central to the productive use of a computer, there is a good complement representing various addressing modes. The first group to be presented compares the contents of the accumulator to locations in memory or to other CPU registers:

Mnemonic	Code
CPA #nn	B7
CPA (U)	27
CPA (X)	07
CPA (Y)	17
CPA nnnn	A7
CPA UH	A6
CPA UL	26
CPA XH	86
CPA XL	06
CPA YH	96
CPA YL	16
CPAI (X)	F7

The last instruction in this group is another

directive that performs automatic incrementing of a data pointer register. It first compares the contents of the accumulator against the location in memory that is *pointed to* by the contents of the 16-bit X register. The address in the X register is then advanced by a count of one. This capability is highly useful when it is necessary to scan a portion of memory while searching for a particular byte of data.

Another group of compare directives provides for the ability to match values in the various CPU registers against immediate data values:

Mnemonic	Code
CPUH #nn	6C
CPUL #nn	6E
CPXH #nn	4C
CPXL #nn	4E
CPYH #nn	5C
CPYL #nn	5E

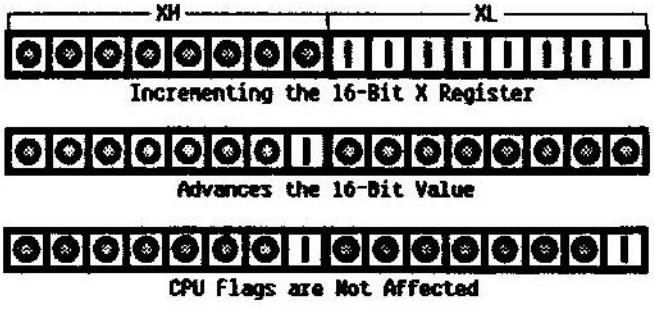
These perform in essentially the same manner as the previous group, with results of the virtual subtraction being used to set up the carry and zero flags. The actual values in the CPU registers (the "compared" values) are not altered.

Increments

The ability to increment the value in a register is important for several reasons. One is so that a value serving as an address pointer can be updated to point to a new location. Another is so that the register can serve as a tally keeper or *counter*. The use of counters is particularly valuable when forming what are known as *program loops*.

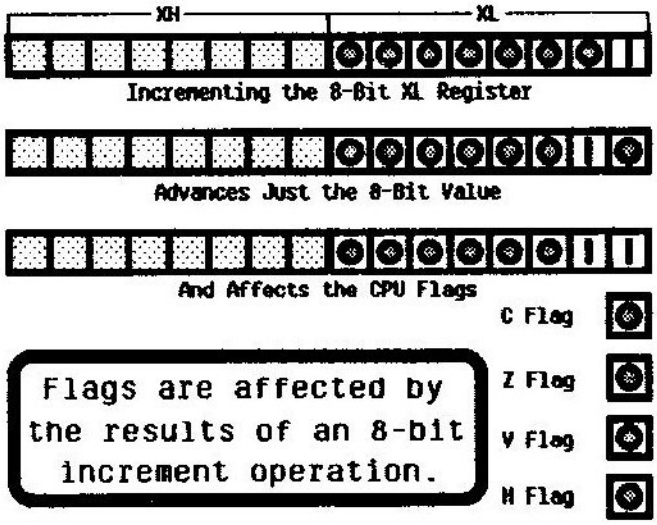
The LH5801 has two basic types of instructions that are used for incrementing values. One type is meant to be used to advance 16-bit data pointer registers. Instructions in this category do not affect the status of the CPU flags.

Mnemonic	Code
INU	64
INX	44
INY	54



The other type is used to advance the count in the accumulator or an 8-bit register (UL, XL or YL). This kind of increment directive will influence the condition of the C, Z, V and H flags.

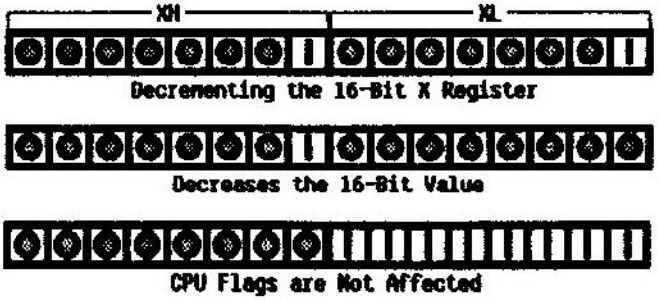
Mnemonic	Code
INA	DD
INUL	60
INXL	40
INYL	50



Decrements

The ability to decrement the values in pointer registers and counters is also valuable. A similar set of directives provides this type of capability in the LH5801. The first subgroup is for decrementing a 16-bit data pointer. None of the flags are affected by the execution of these codes.

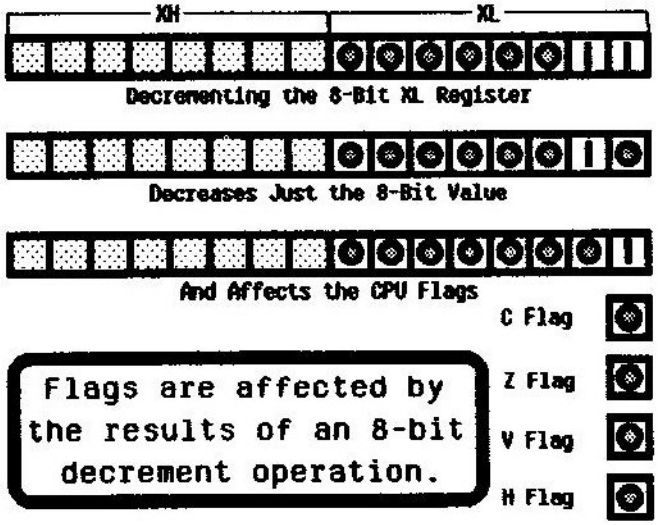
Mnemonic	Code
DEU	66
DEX	46
DEY	56



Mnemonic	Code
DEA	DF
DEUL	62
DEXL	42
DEYL	52

The second group decrements an 8-bit register, either the accumulator or UL, XL or YL. Execution of any of these will affect the status of the C, Z, V and H flags. They are set or cleared as appropriate

according to the result contained in the affected register after the decrement has taken place.



Shifts

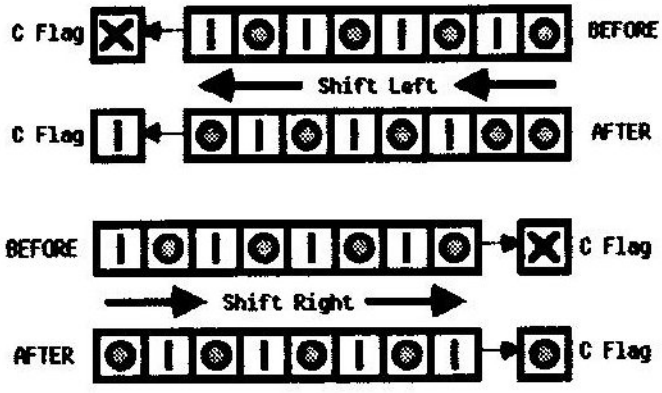
It is often desirable to be able the shift the contents of a register to the right or to the left. The ability to shift data has many uses. Those people who have had experience with digital electronic circuits know this well. Those who have not will soon find it out if they continue to delve into machine language programming.

The LH5801 CPU only has two instructions that are specifically designed for shifting bits. These directives operate directly on the contents of the accumulator. If you ever want to shift the contents of some other CPU register or memory location, you will have to design a little program routine: load the contents of the desired register into the accumulator, shift the contents once they are in the accumulator, then store the result back into the original register!

The two shift directives provided in the LH5801 instruction set are:

Mnemonic	Code
SLA	D9
SRA	D5

The first is used to shift data to the left within the accumulator. The second shifts it to the right.



When a shift occurs, the CPU will always see that the bit at the end of the register (that is being vacated) is cleared to a logic zero regardless of its previous state. On the other end of the register, the bit being kicked out is fed into the carry flag. Thus, you can determine the value of each bit as it is shifted out by examining the status of that (C) flag.

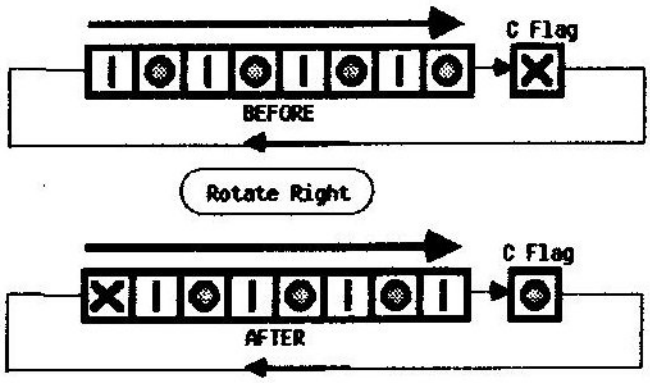
Did you know that shifting a binary value one bit position to the left will multiply it by two? And, yes, shifting it to the right will divide it by two. As you might begin to surmise, the shift operation is a powerful capability to have available when dealing with higher mathematical procedures, such as multiplication and division. It is also handy whenever one wants to deal with data on a bit-by-bit serial basis.

Rotates

If you perform more than one shift operation on the contents of the accumulator, you will lose some data. (Remember, after the first shift, the bit that was shifted out of the register would be temporarily held in the carry flag. As soon as some other instruction that affects the status of the carry flag is executed, including another shift directive, that bit information will be lost.)

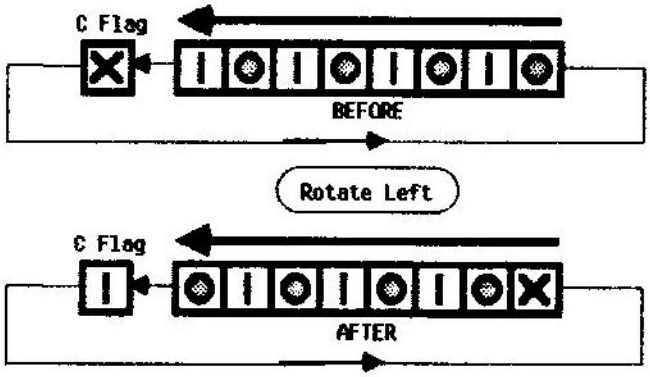
In some situations it may be important to retain all of the information that was originally in the A register, but to reposition that data. Two special directives known as *rotate* commands enable the CPU to perform such a feat.

A register rotate directive is really just a shift with an added feature: the bit shifted out of the register is placed into the carry flag *and* the bit that was originally in the carry flag is fed into the other end of the register. Thus, the bits in the accumulator, coupled by the bit in the carry flag, form a continuous ring.



Rotating to the right results in the carry flag receiving the *least* significant bit from the register. As this is done, the original contents of the carry flag are placed into the *most* significant bit position of the accumulator. The remaining bit positions are all shifted one cell to the right.

Rotating to the left results in the carry flag receiving the *most* significant bit from the register. As this is done, the original contents of the carry flag are placed into the *least* significant bit position of the accumulator. The remaining bit positions are all shifted one cell to the left.



As in the case of the shift directives, there are only two rotate instructions: one to rotate to the left and one to the right. And, the rotating may only be done in the accumulator. You have to bring the contents of other registers into the A register if you want to spin the bits around. Here are the mnemonics and machine codes for the two rotate commands:

Mnemonic	Code
RLCA	DB
RRCA	D1

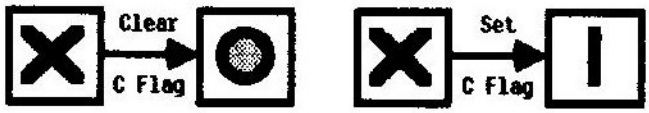
Please take note of the fact that the initial contents of the carry flag will affect what gets rotated into the accumulator by a rotate directive. You should know what the state of the C flag is before executing such an instruction or else the algorithm being utilized should take account of any possible ambiguity in the state of that flag.

Manipulating the Carry Flag

From time to time I have pointed out that it is often crucial to know the status of the carry flag before executing certain types of instructions. This is particularly true for rotates, and those add and subtract commands that utilize the initial contents of the carry flag.

It is possible to place the carry flag into a known state by executing directives using operands that obtain the desired results. For instance, if the value zero is loaded into a register and then an add-immediate directive is used to add the value zero to that quantity, the carry flag would be cleared. There would be no carry from such an addition operation and the carry flag would reflect that fact. Of course, having to resort to such a procedure every time a programmer wanted to make sure that the carry flag was cleared would be a bit cumbersome. It would also eat up memory as a lot of extra data-containing directives might have to added to a program just to manipulate the flags. On the other hand, it is beneficial to remember that in many programming sequences, it will be possible for the programmer to "know" the status of a flag or "force" various flags to a desired state. In such cases, it is possible to save a byte by not having to call on either of the next two instructions to be presented.

Because the carry flag is so central to many kinds of operations, the instruction set does include two directives that will set it to either possible state: *cleared* (to the logic 0 state) or *set* (to a logic 1 condition).



Mnemonic	Code
CLRC	F9
SETC	FB

Whew! Doesn't that take a load off your mind?

Time to Review

So how do you feel? Do you realize that you have already learned about more than *one hundred and thirty-five* instructions? I hope you feel at least a bit proud!

At this point you should spend some time going over what you have learned. Try doing some actual experimenting with the various commands. You can use POKE directives to tuck small routines into memory. If you want to execute those routines from BASIC using a CALL statement, then be sure and terminate the instruction sequence with the code 9A. This will direct the CPU to go back to the BASIC interpreter when it has finished executing the machine language routine.

In the next installment you will learn about the classes of instructions that provide control and decision-making capability to a computer. Indeed, you will be introduced to much of the rest of the instruction set used in the PC-1500 and PC-2.

After that it will be on to presentations of how various instruction sequences may be combined to perform practical functions. We will start building up a library of *utility* routines that you can call upon. You will also learn how to adapt those routines to serve your own special purposes.

You are well on your way to harnessing the *real* power of your pocket computer!

P.O. Box 232, Seymour, CT 06483

 Copyright 1983 POCKET COMPUTER NEWSLETTER

MACHINE LANGUAGE PROGRAMMING

SHARP PC-1500 & RADIO SHACK PC-2 POCKET COMPUTERS

© Copyright 1983 POCKET COMPUTER NEWSLETTER

MACHINE LANGUAGE PROGRAMMING

THE SHARP PC-1500

AND RADIO SHACK PC-2

POCKET COMPUTERS

It might be interesting to point out that all of the instructions discussed in the preceding section have been directives that operate on data. A programmer could organize a series of those instructions in sequential locations in memory. When told to do so, the CPU would proceed to execute each instruction. This sequential type of operation would continue for as long as there were valid instructions to be performed. The reader knows that as each instruction was dismissed with, the program counter would advance over the appropriate number of bytes to point to the next opcode. Typically, this incrementing of the address contained in the program counter is all that is needed to guide the operation of the CPU. That address value always tells the CPU where to obtain the next byte of information to be processed.

Ah, if only life could be so simple and straightforward! Take a step, advance a pointer, know where to obtain the next directive. Beautiful. But it has practical limitations. There can be no freedom of choice or ability to make decisions in such a scheme.

To empower a computer with the ability to alter its course of operation as it evaluates data, there must be an alternative to blindly following a set series of instructions. What controls the sequence in which instructions are executed? The program counter. How can this sequence be altered? By modifying the contents of the program counter!

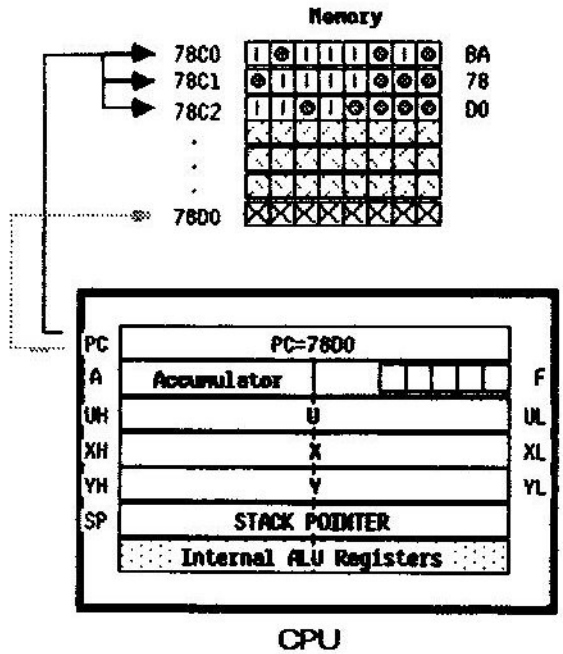
Consider what happens, for example, when a computer is started by applying electrical power. Special circuits "force" an address value into the CPU's program counter. This "tells" the CPU where to begin executing instructions in memory. If it were not for such foresight on the part of designers, the machine would theoretically start trying to execute instructions from whatever memory location randomly appeared in the program counter as power was applied. The result would likely be chaos.

Just as it is useful to be able to load the program counter with a value when a computer is started, it is also useful to be able to tell the computer to switch to another sequence of directives *whenever a programmer desires!* Again, how is this done? By simply altering the value in the program counter so that it no longer blindly points to the next sequential memory location.

A type of instruction that will perform this task is called a *jump* command. It may be used to have the computer skip over a block of instructions or to jump to a particular series of directives.

The fundamental jump directive in the LH5801 has the following mnemonic and opcode:

Mnemonic	Code
JMP nnnn	BA



Note that the instruction utilizes the addressing mode that is categorized as *direct*. That is, the opcode must be followed by two bytes of information that provide the address to which the program is to jump. Those two bytes hold the new address that is to be loaded into the program counter. *That is all this instruction does.* It does not alter the status of any of the flags. It simply causes the computer to abruptly stop executing instructions in one part of memory and start executing them in another area.

There is, however, one very important thing to remember when using the jump instruction. The address specified as the "jump to" location must contain the opcode of another instruction. That is, it must contain the first byte of code for another valid machine language command. Failure to comply with this requirement has been the woe of many an unfortunate programmer!

Artificial Intelligence

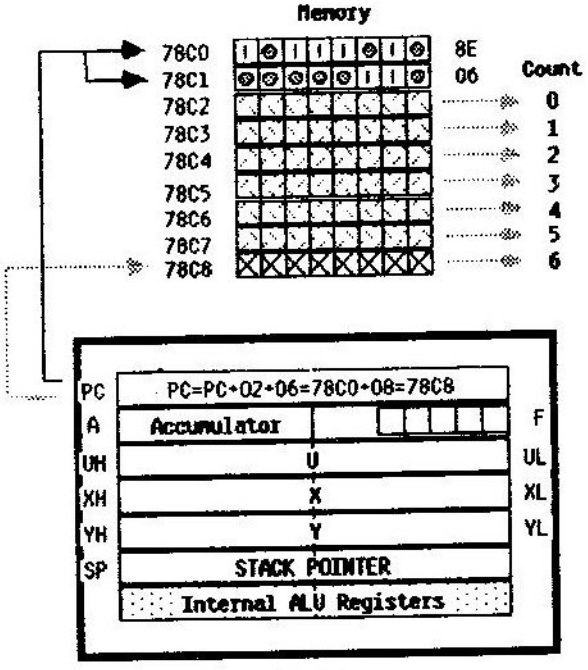
The concepts underlying the group of instructions that will be presented now are some of the most exciting in the computer world. It is this group of instructions that give a computer the attribute of being able to make decisions *and to modify its behavior as a result of those choices!* They form the basis for theoretical models that may someday lead to computers exhibiting actual artificial intelligence. Some people argue that rudimentary models of such behavior are already possible. Read on and learn how a computer mimics this power.

The LH5801 CPU has an instruction that may be considered as a variation of the jump directive. It is referred to as a *branch* command. The essential difference between it and a jump directive is that it only takes up two bytes of memory instead of three. One byte for the opcode, a second for a *relative offset* value. What is this relative offset value? It is the number of bytes, forward or backward, that the CPU is to skip before starting to execute instructions again!

Of course you can see right away that this is in effect a jump directive that is limited in scope. It is limited because the one-byte offset value means that the number of locations that can be skipped is restricted to the maximum value that can be represented in an 8-bit register. (Remember, the opcode for a true jump directive is followed by a two-byte address that can specify *any* location within the 64 kilobytes of memory that could theoretically be assigned to a LH5801 processor.) It turns out, however, that in practical applications, most jumps are relatively short -- they only need to skip over a few instructions -- *so it is worthwhile in terms of memory conservation* to use a two-byte instruction and *branch* to a new, relatively close location, rather than always having to use a three-byte jump directive. This will become more apparent as you gain machine language programming experience.

How does the branch directive actually accomplish its goal of skipping over a block of memory? You guessed it. It simply adds (or subtracts) the relative offset value to the current value in the program counter, thereby changing the program counter to point to a new location!

The key to the successful use of branch directives is to thoroughly understand how to specify the number of skipped bytes. When doing a *forward branch* you start counting with the memory location that immediately *follows* the two-byte branch directive. But, you start counting at *zero!* For instance, if the forward branch opcode (8E) was stored at address 78C0 (hexadecimal, remember) with the offset value in 78C1, and the program was to skip over 7 bytes of code, so that the next instruction executed was at location 78C8, then the offset value would be specified as 6



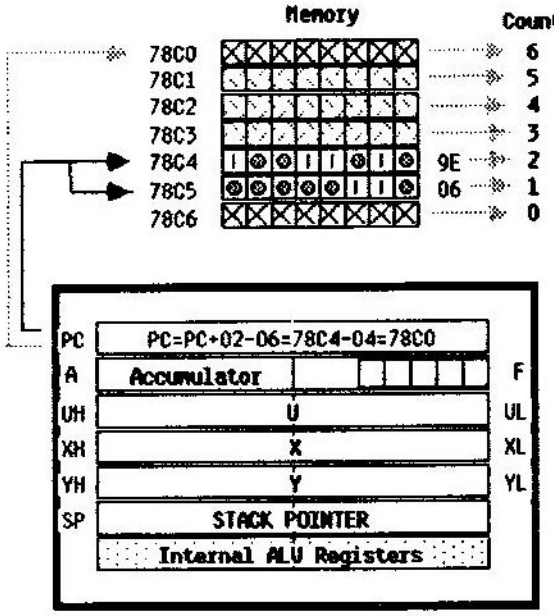
CPU

(actually, 06 using hexadecimal notation). Note that the offset value specified is six even though seven bytes of code are being skipped. This is because the first skipped byte is numbered zero. Do you know why this convention is used? It is because the program counter will already have advanced to the next byte of memory by the time it is ready to implement the offset value! Makes sense, right?

Be careful when it comes time to specify a *reverse branch*. You can still start counting at the byte following the offset value, but you must again count this as the zero byte and you progress backwards (towards lower memory addresses) from that point. Thus, if the reverse branch opcode (9E)

 Copyright 1983 POCKET COMPUTER NEWSLETTER

was stored at location 78C4, the offset value was in 78C5, and the program was to go back to an instruction that started at location 78C0, then the offset would again be specified as 06.



CPU

It is absolutely essential that you understand how to specify these offset values in order to effectively apply these most valuable directives! Do not leave this section until you have carefully studied this text and the accompanying diagrams and are confident you understand the concepts.

Perhaps one way to enhance your understanding is to recall how these directives affect the program counter: the offset value is either added to or subtracted from the program counter, thereby immediately causing it to point to the new location at which to find the next instruction. Now, what value will the program counter have in it at the time it finishes executing a branch directive? Why the address of the byte that follows the current instruction. This has to be the case because the program counter is always telling the CPU where to obtain its operational information.

In the case of a forward branch, advancing the value in the program counter by whatever number of bytes the programmer wants to advance, less one to account for the byte assigned the count of zero, will cause it to be pointing to the desired location. This is easily demonstrated by assuming that rather than skipping anything, the CPU is to simply execute the next instruction stored in memory. In this case the branch offset value would be zero. It would be exactly the same thing as though the program counter had merely advanced itself to the start of the next instruction, as it normally does, when it had finished executing the instruction! Naturally the FB 00 (forward branch with a zero offset value) directive, while it can be specified, has little practical value. The example may, however, help illustrate how the instruction works.

In the case of the reverse branch, where you want to go back a certain number of steps, you have to remember that the program counter is already pointing to the next instruction it would normally execute if the branch did not occur. The offset value will now be *subtracted* from the value in the program counter. If the count of two is subtracted from the program counter at this point, what will happen? Right. The program would instantly be caught in an endless loop. This is because the hapless program counter would be reset to point to the opcode of the reverse branch instruction itself. It would repeatedly perform the same directive over and over again. Remember this well: *a reverse branch with an offset value of two is a sure way to produce CPU lockup!*

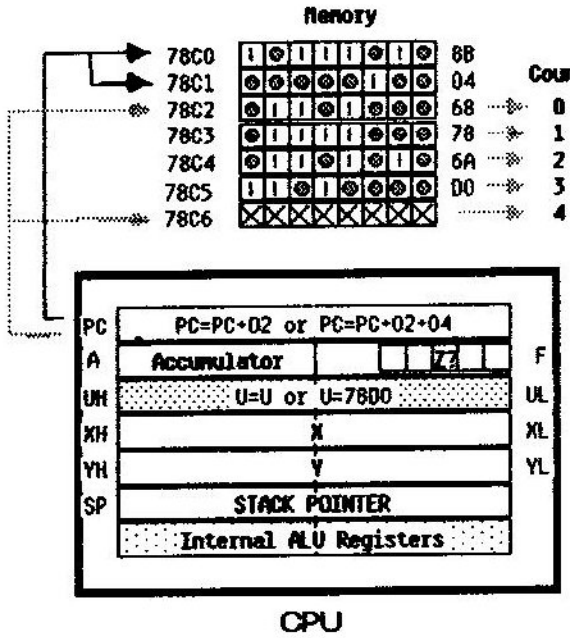
In summary, a plain forward or reverse branch is merely a shortened form of a jump directive. It simply uses *offset addressing* rather than direct addressing. It is, however, the many various forms of the forward and reverse branching instructions that give the LH5801 CPU its decision-making capability.

Flagging a Choice

Remember all those CPU flags we discussed earlier? Now is when you learn how essential they are in controlling the operation of a computer!

Those flags can be "tested" by instructions, such as the branch group being presented here. That is, as part of the operation of a particular instruction, the status of a flag can be examined to determine its current state. Then, depending on the state found, the final operation of the current instruction can be altered. In the case of the branch directives, the alteration to the operation is as follows: If a tested flag is in a desired state (the test condition is *true*), then the branch procedure will take place, *otherwise it will not!* Naturally, if a particular branch instruction is not executed, that is, if the branch (jump) does not take place, then the instruction that follows the branch will be executed *instead of being branched (jumped) over!* Presto! You now have the capability of changing the path of instructions that will be followed *depending on the flag conditions that exist at the time a program is actually being performed.* Great stuff!

Take, for illustration, the *conditional forward branch if zero* directive (represented by the mnemonic FBZ). Again, purely for illustration, let us suppose that the offset value accompanying the opcode was 04, so that the computer would branch ahead *over* the four bytes (to the *fifth* byte) if the



CPU

command was executed. And, to complete the picture, imagine that the instructions LDUH #78 and LDUL #D0 (which are each two-byte directives) followed the branch directive. What would happen as this series of commands was encountered by the CPU?

The FBZ #04 would be encountered. The CPU would examine the status of the zero flag. If it was *set (true)* then a count of four would immediately be added to the program counter. *This would cause the computer to skip over the next four bytes of information in memory and continue executing instructions at the fifth byte.* Thus, the LDUH #78 and LDUL #D0 directives would *not* be performed *if, and only if,* the zero flag was found to be set at the actual time that the branch instruction was executed. On the other hand, if the CPU found that the zero flag was *cleared (false)* at this time, then *the branch would not take place.* Instead, the program counter would merely advance in a normal fashion thereby causing the LDUH #78, LDUL #D0 directives to be performed.

You have now seen how a branch directive may be used to cause something to be done or not done. In this example, the 16-bit U register would be set to point to address 78D0 if, and only if, the zero flag had been set when the branch directive was encountered. Otherwise, whatever previous value had been in the U register would be left there.

The converse instruction is also available as part of the LH5801's repertoire. That is, there is a directive: *forward branch on non-zero* (FBNZ). This instruction is exactly the opposite of the one just described. If the zero flag is cleared (i.e., the register that controlled the flag was non-zero), then the forward branch *is* performed. If the zero flag is set, then the forward branch is *not* taken.

There are six other conditional forward branch instructions that perform similarly, based on the status of the carry (C), half-carry (H) and overflow (V) flags. Thus, there are a total of nine variations of the forward branch directives: one *unconditional* and eight *conditional.*

Mnemonic	Code
FB #nn	8E
FBZ #nn	8B
FBNZ #nn	89
FBC #nn	83
FBNC #nn	81
FBH #nn	87
FBNH #nn	85
FBV #nn	8F
FBNV #nn	8D

Similarly, there are a total of nine variations of the *reverse* branch directives: one *unconditional* and eight *conditional.*

Mnemonic	Code
RB #nn	9E
RBZ #nn	9B
RBNZ #nn	99
RBC #nn	93
RBNC #nn	91
RBH #nn	97
RBNH #nn	95
RBV #nn	9F
RBNV #nn	9D

Note that the only difference between the forward and reverse branch directives is the direction in which the jump occurs. Forward branches move ahead in memory (resulting in a higher address value in the program counter). The reverse branches move backwards in memory (resulting in a lower address value in the program counter).

(Readers who are planning on becoming MLP Masters might also note that the only difference in the opcodes is that the reverse group turns on one more bit. Thus the most significant digit in the opcode becomes a 9 instead of an 8.)

Having such a complete set of conditional branch directives eases the task of developing machine language programs. Note, for instance, that having both branch on zero and branch on non-zero directives, is really a convenience. A programmer could obtain the same logical result in a program if only one of those directives was available, through proper positioning of the code that was to be skipped. Having both possibilities present as instructions, however, makes life easier for the programmer. Say "thank you" to the chip designer, please.

 Copyright 1983 POCKET COMPUTER NEWSLETTER

See for Yourself

Let's examine a little program that demonstrates the use of branch directives. Actually this example is a modification of a simple routine that originally appeared in the instruction manual that is supplied with the Radio Shack PC-2. The machine language routine to be presented will cause whatever is in the liquid-crystal display (LCD) to be displayed in inverse format. A flashing display is then made by coupling the machine language directives to a simple BASIC program.

Here is the series of machine language instructions that will be utilized:

Mnemonic	Description
LDXH #76	Load X-high with value 76.
LDXL #00	Load X-low with 00.
LDA (X)	Load accumulator with the contents of the address pointed to by register X.
EORA #FF	Exclusive OR (invert) the bits in the accumulator. (A commonly used programming "trick".)
STAI (X)	Store the inverted value in the address pointed to by X, then automatically increment the value in the 16-bit X register so that it points to the next location in memory!
CPXL #4E	Compare the contents of X-low with the hexadecimal value 4E.
RBNZ #08	If the result of the comparison is non-zero (i.e., X-low is not equal to 4E), *reverse branch* back to the LDA (X) command.
CPXH #77	Compare the contents of X-high with the hexadecimal value 77.
FBZ #04	If the result of the comparison is zero (i.e., X-high is equal to 77), *forward branch* ahead to the RTS command.
LDXH #77	Load X-high with 77.
RB #12	*Reverse branch* to do the LDXL #00 instruction again and go through second half of LCD buffer area.
RTS	Special command that will pass control back to BASIC program.

The machine language routine as just presented is in what is referred to as *assembly language* or mnemonic form. This is also sometimes referred to as a *source code listing*. Note that there is no indication of where the instructions will be stored in memory within the computer nor is the actual opcode used by the CPU even presented. Instead, just the mnemonic representations for the instructions that will be used are listed. This is the method of notation that experienced machine language programmers use as they think about and create routines that will ultimately be executed by the CPU.

Before such a listing can be used by a computer such as the Sharp PC-1500, the assembly listing must be converted to what is known as the machine readable or *object code* form. This process is also known as *assembling* a program. It consist of nothing more than translating the mnemonic representations for the instructions into the actual binary patterns recognized by the CPU and assigning them to specific memory locations (addresses) in which to reside.

The process of assembling a program is readily accomplished using manual methods, particularly when the program is small. All that is required is a lookup table that gives the machine code for each type of instruction that is being invoked. It may also be necessary to ascertain specific addresses, such as when data is being stored or jump instructions are being used. And, in the case of branch directives, it is necessary to calculate offset values. However, it should be noted that all of these processes are essentially mechanical in nature. That is, a computer can readily be instructed as to how to accomplish such translations and perform address calculations. Hence, it should come as no surprise that it is possible to construct an *assembler program* that will process a mnemonic source listing and convert it to a final object code listing. Professional machine language programmers who work at developing machine language routines on a continuous basis almost always use such a program. It relieves them of the tedious task of performing such translations manually.

Alas, students using this tutorial should plan on becoming adept at using manual assembly methods. The routines presented herein will readily yield to such rudimentary methods while providing many useful insights. The development of an assembler program for the LH5801 CPU that would run on a PC-1500 could easily take several person-months to develop, would use a substantial amount of memory, and would likely find a rather limited market.

Here is how the above routine might appear in "assembled" form:

Address	Code	Mnemonic
78C0	48 76	LDXH #76
78C2	4A 00	LDXL #00
78C4	05	LDA (X)
78C5	BD FF	EORA #FF
78C7	41	STAI (X)
78C8	4E 4E	CPXL #4E
78CA	99 08	RBNZ #08

```
78CC    4C 77     CPXH #77
78CE    8B 04     FBZ #04
78D0    48 77     LDXH #77
78D2    9E 12     RB #12
78D4    9A        RTS
```

Does it make sense to you? I hope so! Note that we have (arbitrarily) decided that the code will be stored in memory starting at address 78C0 (using hexadecimal notation). Do you recall that memory locations 78C0 through 78CF are normally used to store the string variable A$ by the PC-1500 BASIC interpreter? Locations 78D0 through 78DF are normally used to store the string variable B$. Do not use these variables while this routine is residing in memory!

If you have Norlin Rober's LMD program (as recommended in the preceding section of this tutorial), you can use the monitor portion to load the above code directly into memory. If this is the case, then you can skip the first three lines of the following BASIC program.

```
1000: POKE &78C0,&
      48,&76,&4A,0
      ,5,&B0,&FF,&
      41
1010: POKE &78C8,&
      43,&4E,&99,8
      ,&4C,&77,&8B
      ,4
1020: POKE &78D0,&
      48,&77,&9E,&
      12,&9A
1030: INPUT "YOUR
      MESSAGE?",X$
1040: CALL &78C0
1050: FOR X=1TO 50
      :NEXT X
1060: GOTO 1040
```

Note that the first three lines of this BASIC program store the desired machine codes into memory. The remaining lines are used to demonstrate the action of the machine language routine. If you do not use lines 1000 - 1020 of this BASIC program (because you use a monitor to load in the machine codes) then *make sure you do not use a RUN command to execute the program!* If you do so, you will simply *wipe out* the machine codes that were stored where variables A$ and B$ are kept by BASIC! This is because BASIC clears out (initializes) those locations whenever it is given a RUN command. Instead, use a GOTO directive -- such as GOTO 1030 -- to execute the BASIC program without initializing the BASIC variables.

A few words as to what the machine language routine does is undoubtably in order at this point.

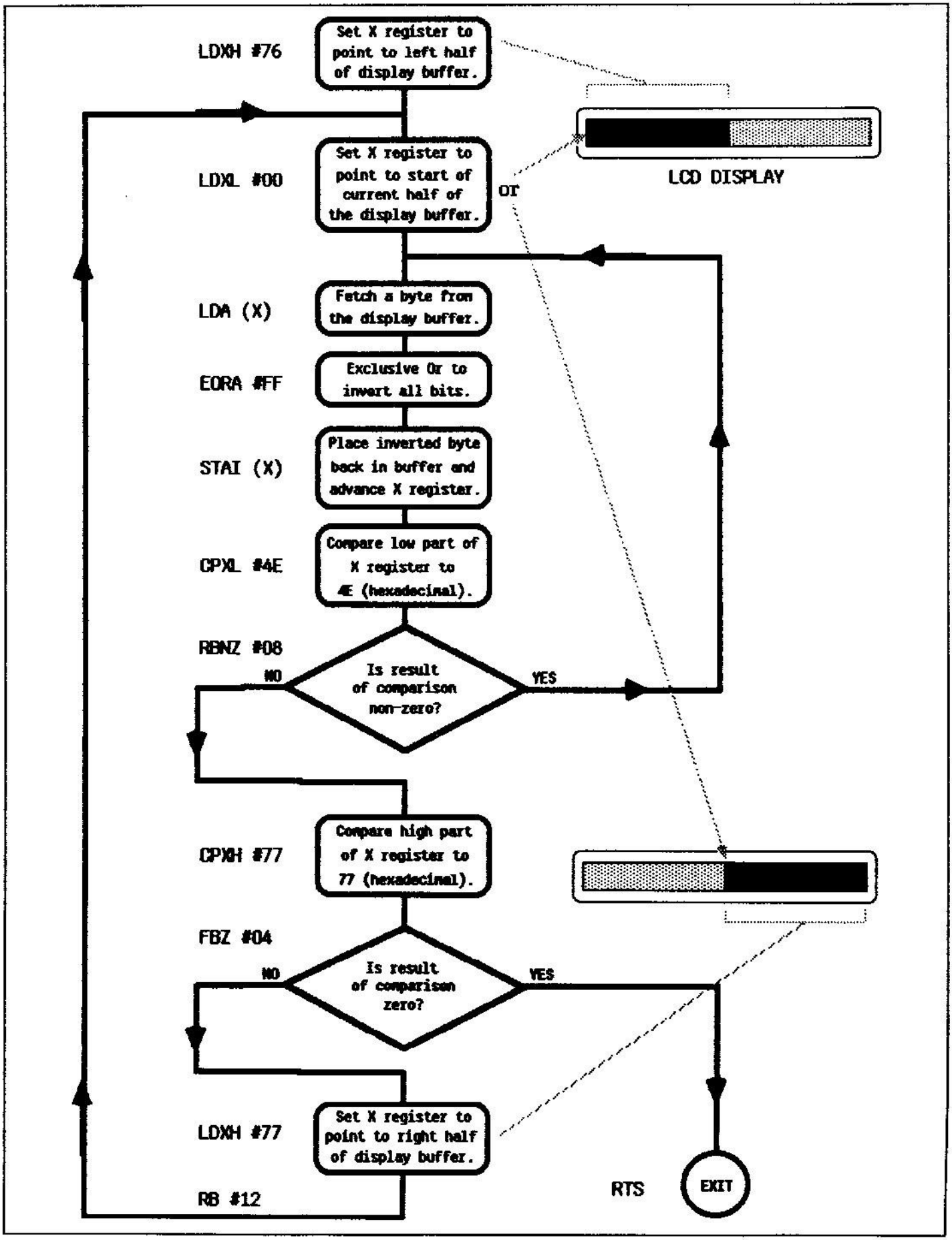
You see, the PC-1500 maintains a display "buffer" (a holding zone) in locations 7600 through 764D and 7700 through 774D in memory. This effectively splits the LCD into left and right halves with each block of memory serving one half of the display. (This is a simplification. The addressing of the display is actually more complicated. But, considering it as consisting as right and left halves will do for the sake of this discussion.) The machine language routine being demonstrated does the following:

It sets the address 7600 (the start of the left half of the display buffer) into the 16-bit X register of the CPU. It then fetches a byte of data from the address pointed to by the contents of the X register into the accumulator. That byte of data is then *inverted* using an exclusive OR instruction "against" a byte where all the bits have been set to a logic one state. The inverted data is then stored back into the same memory location in the display buffer. However, the STAI (X) instruction, you may recall, automatically increments the contents of the X register so that it points to the next location in the display buffer (in this case).

Since the first (left) half of the display buffer ends at address 764D and we do not want to alter memory beyond that location, a test is used to determine when that address has been exceeded. This is accomplished by comparing the lower byte of the X register against the value 4E (the value immediately after 4D). This is the value that the lower part of the X register will contain as soon as the STAI (X) instruction has finished executing (when the X register contains the address 764D). Remember that a compare directive simply sets the CPU flags based on the results of the compare. Thus, if the comparison of X-low against the value 4E is non-zero, then the CPU "knows" (in this situation) that the X register has not reached the value 764E. The RBNZ #08 directive thus routes the CPU via the program counter back to execute the LDA (X) directive (at address 78C4 in the assembled listing). A program loop has effectively been formed. This loop or series of instructions will be repeated until X-low attains the value 4E. At that point, with the comparison being equal to zero, the reverse branch will not take place. Instead, the CPXH #77 directive (at address 78CC) will finally be executed.

This compare determines if the entire buffer has been processed (i.e., both halves) by checking to see if the high portion of the 16-bit X register is set to the hexadecimal value 77. If this is indeed the case, then the routine is finished. A forward branch (FBZ #04) is used to direct the program to a special instruction (yet to be introduced) that will conclude the operation. However, if X-high is not yet 77 (in which case it must be 76 as that is what it

 Copyright 1983 POCKET COMPUTER NEWSLETTER


LDXH #76
Set X register to point to left half of display buffer.
LDXL #00
Set X register to point to start of current half of the display buffer.
OR
LCD DISPLAY
LDA (X)
Fetch a byte from the display buffer.
EORA #FF
Exclusive Or to invert all bits.
STAI (X)
Place inverted byte back in buffer and advance X register.
CPXL #4E
Compare low part of X register to 4E (hexadecimal).
RBNZ #08
NO
Is result of comparison non-zero?
YES
CPXH #77
Compare high part of X register to 77 (hexadecimal).
FBZ #04
NO
Is result of comparison zero?
YES
LDXH #77
Set X register to point to right half of display buffer.
RB #12
RTS
EXIT

is initially set to by the program), then it must be set to that value (77) so that the right half of the display buffer can be processed. When that is the case, then the RB #12 (reverse branch) instruction is used to send the program back up to the LDXL #00 instruction at address 78C2. This resets the lower part of the X register too, so that the region from 7700 through 774D can now be processed by the same series of instructions that originally did the processing when the X register held values in the range 7600 - 764D!

So what do you think? There is a little machine language routine with barely a baker's dozen directives and yet look at all the work it is doing! And look at all the details involved. Is it worth learning how to do these types of maneuvers?

To summarize, that little routine essentially scans the contents of the display buffer. As it obtains each byte, it inverts all the bits. Those bits determine whether each dot making up a single column in the display is on or off. Thus, the operation results in the display being switched from normal to inverse mode. (The process is somewhat complicated by the fact that the actual display buffer is spread over several separate sections of memory. Such inconveniences frequently occur in the world of machine language programming.)

To make a flashing display (that alternates between normal and inverse modes), the machine language routine is coupled to a simple BASIC program. (The resulting combination will be referred to as a *hybrid* program, in that it contains both BASIC and machine language coding.) This BASIC program (besides poking the machine codes into memory) allows the user to define a message. (That is, set up the display buffer with something nifty for demonstration purposes.) It then uses the machine language routine to invert the display. This inversion will take place in the blink of an eye because the machine language routine is executed in just a few milliseconds. Next, a BASIC loop is used to insert a time delay. The program then jumps back to invert the display again. This causes an alternating or flashing effect. If the time delay (line 1050 in the BASIC program) was not provided, the program would perform so fast that you would barely be able to see the alternating effect. (You can experiment with the rate of flashing by changing the "TO" value in line 1050.)

Say, have you noted that a machine language branch directive is the equivalent of a GOTO statement in BASIC? Good.

The Super Looping Branch

Ah, the prowess of chip engineers! The game of one-upmanship continues even in the world of microcomputers. The LH5801 CPU can execute a very special type of branch instruction that has special application in the area of forming program loops (a section of a program that is repeated a set number of times). This instruction has the following mnemonic and machine code:

Mnemonic	Code
BNZD #nn	88

The mnemonic stands for *branch on non-zero and decrement*. What this instruction does is test the value in the 8-bit UL register. If it is non-zero, then the value in the immediate byte that follows the opcode will be *subtracted* from the program counter. If it is zero, then the program counter is advanced in the normal fashion so that the next instruction stored in memory will be executed. Regardless of the results of the test of the UL register, the contents of that register will also be decremented. Note that the decrement takes place *after* the test has taken place. Note also, that while the test is for a zero condition in the UL register, this test does *not* affect the status of the zero flag (or any other regular CPU flag).

To see the practical application of this special type of branch directive, suppose you wanted to perform a particular sequence of instructions five times. To use the BNZD instruction, you would first load the value 04 into the UL register. (Can you recall the directive that would accomplish this?) Note that the setting up of the UL register must take place outside of the series of instructions that will make up the program loop. (If this advice is not followed, the computer will get caught up in an "endless loop" situation. This can be most embarrasing for the careless programmer.)

For the sake of illustration, assume the sequence of directives that was to be repeated resided in six bytes. After including the two bytes used by the BNZD directive itself (the opcode and the branch offset value), the proper offset value (immediate data byte following the BNZD opcode) for this example would be 08.

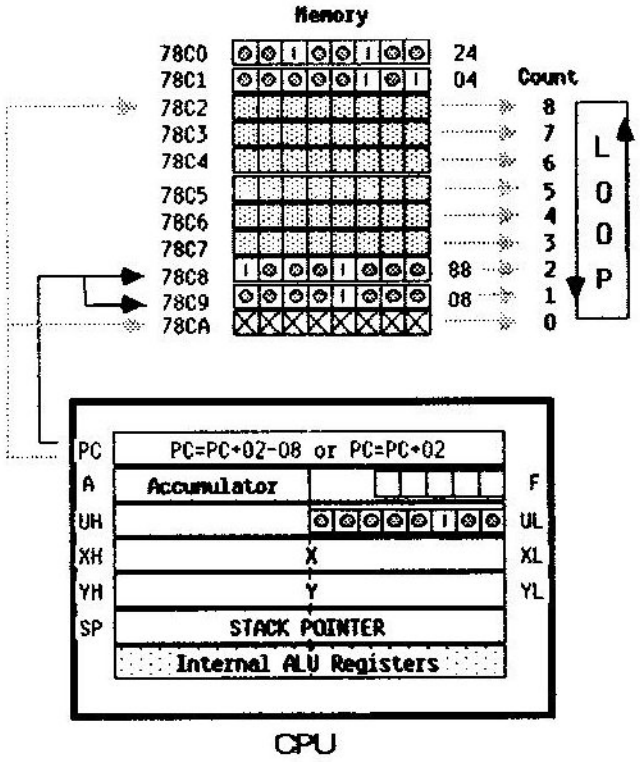
Can you mentally step through the operation of this instruction for the above illustration? Let's do it just to check things out....

The first time the BNZD #08 directive is encountered, the UL register will initially contain the value 04. This is certainly non-zero, so the offset value 08 will be subtracted from the value in the program counter at the conclusion of the operation. This will cause the computer to "loop back" and re-execute the same series of directives starting eight bytes back from where it was at the time it *finished* executing the BNZD command. (Remember, when the CPU has finished executing an instruction, the program counter will have

 Copyright 1983 POCKET COMPUTER NEWSLETTER

automatically been advanced the number of bytes consumed by the current directive. Thus it will be pointing to the address immediately following the two bytes used by the BNZD #08 directive itself.) Also, at the conclusion of this first encounter, the value in UL will be decremented so that it is left with a value of 03.

As the BNZD directive is encountered on the second time through the series of directives, the UL register will contain 03 (still non-zero), thus the reverse branch will again be performed and UL will decrease to a value of 02.



This process repeats until the fifth trip through the "loop" (in this example). This time UL will have a value of zero when the BNZD directive is encountered. This means the branch will *not* be performed. Instead, the program counter will advance normally. The program loop will thus be exited as program execution continues with the opcode contained in the next byte of memory. Note, however, that UL will still be decremented *and will end up with a value of FF!* That is, as the zero value is decremented, the UL register will underflow to the all ones (hexadecimal value FF) condition.

This special looping branch is a powerful directive. The availability of such a command should be considered a luxury by machine language programmers. Earlier types of microprocessors did not have such built-in convenience. You had to roll your own from scratch. (Can you figure out what series of instructions to use to mimic the operation of the BNZD directive?)

The Machine Language GOSUB

Just as you can emulate GOTO statements using jump and branch directives in machine language, there are also instructions that provide subroutine capability at the machine language level. In this tutorial these types of instructions will be referred to as *jump to subroutine* and *call* directives depending on the addressing mode being utilized.

In the BASIC language, a GOSUB statement does the following: the program jumps to a given line number instead of executing the next available statement in the program. However, before skipping off to the designated line, it *saves* its current location (more precisely, the location of the next statement it would encounter had it not performed the GOSUB). It does this so that it can eventually return to resume operations where it had left off before being called away by the GOSUB directive. The return location is saved in a *last-in, first-out (LIFO) stack* that is maintained by the BASIC interpreter.

The program then begins executing a new series of statements starting at the line number indicated by the GOSUB directive. It executes the new series until it encounters a RETURN statement. The RETURN directive causes the last location stored in the LIFO stack to be used as the point to which it is to return. Thus the program "returns" to the point just beyond where the original GOSUB directive occurred and continues its operation.

Machine language programs can also invoke subroutines. If a subroutine is located at a specific memory address, then an all-encompassing *jump to subroutine* instruction may be used to direct the CPU to do the following: save its current location (the address of the next instruction that it would normally execute), then jump to a specific location in memory and begin performing the directives that it finds at that point. How does the CPU do this? You already know that it can jump to a new area in memory by changing the value in the program counter, just as it does for an ordinary jump directive. It can "remember" where it was before going off to perform the subroutine by merely saving the "current" value in the program counter. (The *current* program counter value refers to the value it would have after acting on all the bytes making up the jump to subroutine directive. That is, the address of the byte in memory that follows the bytes used by the jump to subroutine instruction.) Where does it save the value of the program counter? Why in the location pointed to by the contents of the *stack pointer!*

Do you remember what happens when the stack pointer is used by the CPU? Depending on the operation, the stack pointer is incremented or

decremented. When it is used in conjunction with a jump to subroutine directive, it operates as follows: The low portion (least significant eight bits) of the program counter (which is pointing to the third byte of memory after the opcode for the jump to subroutine directive) is stored in memory at the address pointed to by the current value in the stack pointer. The value in the stack pointer is decreased by one. The high portion (least significant eight bits) of the program counter is now stored in memory at the new address pointed to by the stack pointer. The stack pointer value is decremented again so that it is ready to point to the next byte in the current *stack*. In other words the stack is constructed from the "top" of memory downwards. This can be contrasted to the storage of instructions themselves which are executed from the "bottom" of memory upwards. Said another way, the program counter normally advances, while the stack pointer goes backwards as it constructs a stack. (But, it goes forwards when it dismantles a stack!)

Do you understand that a stack in memory created by the operation of the stack pointer can be located in any section of RAM? Did you know that a programmer can create a whole host of stack areas in memory? Do you realize that a program that utilizes the stack pointer must first make sure that the stack pointer is loaded with an appropriate memory address at which to construct a stack? (Yes, there are directives to accomplish this task. They will be presented in due course.) Have you noticed that while the stack pointer seems to operate in reverse, it ends up storing subroutine return addresses in the same order as which they appear when associated with instruction opcodes? That is, the high portion of a two-byte address value ends up in the lower-valued address byte of the stack and the low portion ends up in the stack byte with the higher address.

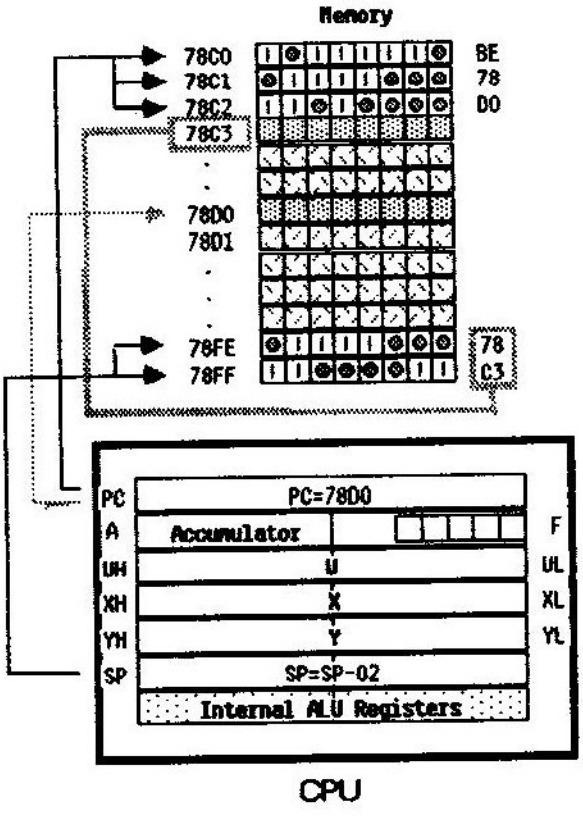
The mnemonic and opcode for the standard *jump to subroutine* directive is:

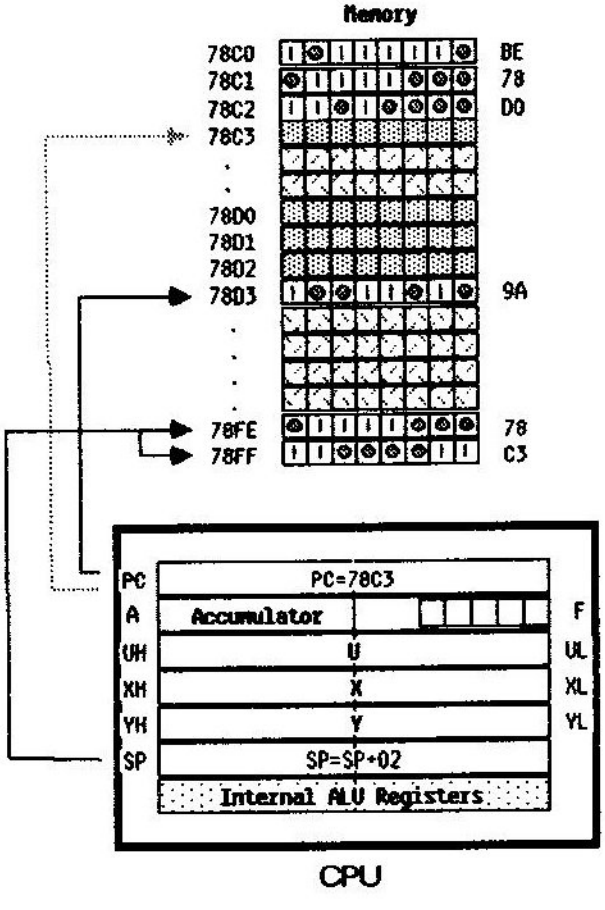
Mnemonic	Code
JSR nnnn	BE

Once the program has jumped to the start of a subroutine, (which must contain a valid instruction opcode), the CPU will continue executing instructions in its normal sequential fashion. It does so until it encounters a special directive that must be used to terminate *all* subroutines. It is known as the *return* instruction. (Wow! Just like in BASIC!) The LH5801 recognizes the following opcode as the standard *return from subroutine* directive:

Mnemonic	Code
RTS	9A

When this command is detected, the program counter is reset to the two-byte address that was saved in the stack when the subroutine was called.





 Copyright 1983 POCKET COMPUTER NEWSLETTER

(Provided that the program has been constructed properly. Read on.)

What actually happens when a RTS directive is found is the following sequence of events: The address in the stack pointer register is increased by one. The contents of the memory location then pointed to by the stack pointer is loaded into the left-most (most significant) half of the program counter. The stack pointer value is increased again by one. The contents of the memory location it then points to is fed into the least significant half of the program counter. The CPU resumes program execution at the address that has just been loaded into the program counter.

Note that if the last thing the stack pointer did previously was to load a two-byte value (address) into a stack in memory (such as occurs when executing a jump to subroutine directive), then a RTS directive will result in that saved address being placed back into the program counter. The stack pointer is also restored to its previous value. The LIFO (last-in, first-out) stack process has been completed.

You might also like to take note of the fact that this orderly process can be subverted by a careless programmer. If for example, the stack pointer is used for some other purpose (and is not properly restored) during the operation of a subroutine, then it will not be able to restore the proper return address to the program counter when an RTS directive is encountered. The execution of a RTS instruction, will however, still cause the program counter to be loaded with whatever two bytes the stack pointer happens to be pointing to. Woe be it to the haphazard programmer who allows such a situation to occur. The result is invariably instant chaos and a totally "bombed" program!

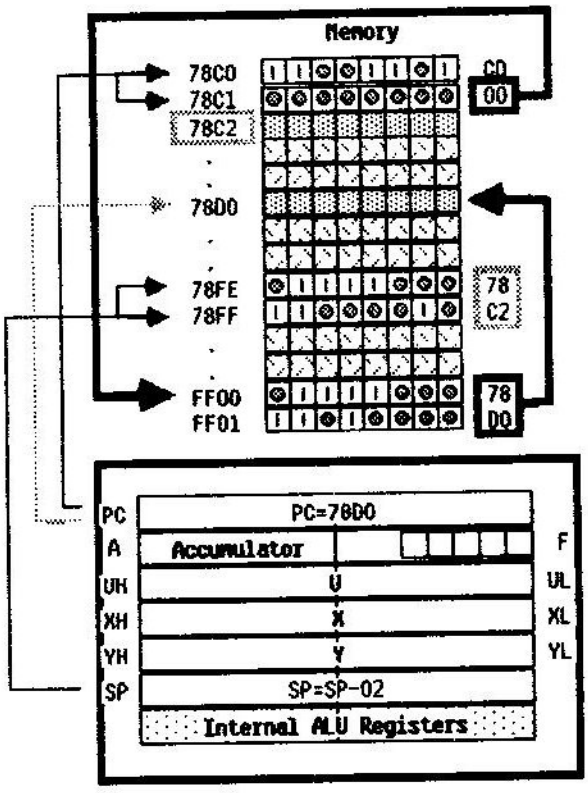
The use of stacks and their control by the stack pointer register is of great importance (besides their use in keeping track of subroutine return addresses). There will be a special section devoted to this particular aspect of machine language programming further on in this series.

Special Subroutine Calling Instructions

The LH5801 CPU can execute several different forms of the fundamental subroutine call directive. Unfortunately, for many PC users, the availability of these instructions will be primarily of academic interest. The reason for this will become clear shortly.

One secondary class of call directives that the LH5801 can perform uses what is referred to as *base page indirect addressing*. Instead of the opcode for the instruction being followed by two bytes containing an absolute address (of where the subroutine starts), it is followed by a single byte. This byte points to the location in a specific block of memory (the *base page*) where a two-byte absolute address value may be found. The base page used by the LH5801 is referred to as page FF. It is the 256-byte block of memory having the hexadecimal address range FF00 through FFFF. Note that it encompasses all the addresses wherein the two most significant digits (FF) of the address do not change, hence the reference to the FF *page*.

In the LH5801 the immediate byte that follows the opcode for this class of call instruction must contain an "even" value, i.e., 00, 02, 04, ... FC, FE. This value provides the location *on page FF* of the first byte (high-order portion) of a two-byte address value. It is this address value to which the CPU is to jump and begin executing a subroutine.



Note that since two bytes are required for each address on page FF and since there are 256 bytes on that page, this type of call directive can only refer to one of 128 different memory addresses stored on page FF at any given time.

Now here is the reason why this type of directive will be of purely educational benefit to many PC users: in the Sharp PC-1500 and Radio Shack PC-2, the address range FF00 - FFFF is occupied by ROM! It is filled with address values that refer to subroutines used by the BASIC package that is similarly installed in ROM. Since these locations cannot be altered by a machine language programmer, they will not be of any use to someone doing machine language programming.... Unless, that programmer can make use of some of

the routines contained within the BASIC ROM itself. (That is not an impossible idea, but it is beyond the scope of this present discussion.)

Never-the-less, this type of call instruction does exist. Indeed, it exists in unconditional and conditional (controlled by flags) form as given in the following compilation:

Mnemonic	Code
CALL #nn	CD
CAZ #nn	C8
CANZ #nn	C9
CAC #nn	C3
CANC #nn	C1
CAH #nn	C7
CANH #nn	C5
CAV #nn	CF

The conditional forms are similar in concept to those described for the conditional branch group. If the flag condition being tested for is met, then the subroutine call is performed, otherwise the CPU ignores the directive and simply performs the next instruction in the current series.

Remember too, that when a call directive is performed, a return address is saved on the stack. The return address is the address of the byte that follows the two bytes used by the instruction (opcode and location on the base page). Study the accompanying diagram to recall, if necessary, how the contents of the program counter are saved at the address pointed to by the stack pointer, just as is done when a directly addressed jump to subroutine (JSR) instruction is performed.

From a practical viewpoint, these two-byte call directives cannot be used unless: (1) You plan to make use of some of the subroutines stored in the pocket computer's BASIC ROM or (2) You plan to unsolder a LH5801 and build your own computer so that you can put the indirect addresses you want in the FF00 - FFFF block of memory!

However, it is worth discussing why these types of instructions are worth having, at least from a programmer's perspective. What is the primary reason for having subroutine capability built into a CPU? It is to conserve on the amount of memory required in a system by permitting the repeated use of selected blocks of code!

As code is developed for a large program, it usually develops that a number of algorithms or procedures are repeated over and over again, but at different points in the program. Instead of having to always insert the same block of code at each place where a duplicate procedure must take place, the experienced programmer will simply place the key series of instructions at a convenient location in memory and terminate it with a RTS (return from subroutine) directive. The starting address of that key block of instructions (subroutine) is then placed as the address bytes of a JSR command whenever it is desired to perform that operation. The net result is the saving of a lot of memory.

Suppose a particular algorithm occupies 50 bytes of program memory. If it were repeated 20 times within a program, it would consume 1000 bytes of storage. Placing the 50 bytes of code into a subroutine (adding one byte for the terminating RTS command) and jumping to it as a subroutine (requiring just 3 bytes of code each time it was referenced) would reduce the memory storage requirement fromm 1000 to just 111 bytes. It is a powerful, essential, programming concept.

The two-byte call directive just described can reduce the overhead even further. As an exercise, try determining how much memory the above example would consume if all references to the subroutine were through a two-byte call directive. (Don't forget the two bytes consumed by the address that would have to be stored in page FF.) Do you come up with a total of 93 bytes?

In fact, the example just given is quite conservative in practice. In a cursory study of the code stored in the ROM used in the PC-1500, it appears that there are between 1,000 to 2,000 references to subroutines within the roughly 16,000 bytes of coding. If it were not for the extensive use of subroutines, the BASIC ROM coding might well occupy 128K of memory! Just the use of the two-byte call directive over the three-byte JSR probably saves about 1,000 bytes of storage. That represents about a six percent reduction in memory requirements.

With that concept in mind -- that the use of subroutines saves a lot of memory space -- you probably will not be surprised to learn that the LH5801 has still another type of jump to subroutine (call) directive. It is a *one-byte* call!

How does it do this? The LH5801 CPU simply recognizes the following opcodes as special call directives using a rather nifty relationship:

Special One-Byte Call Opcodes

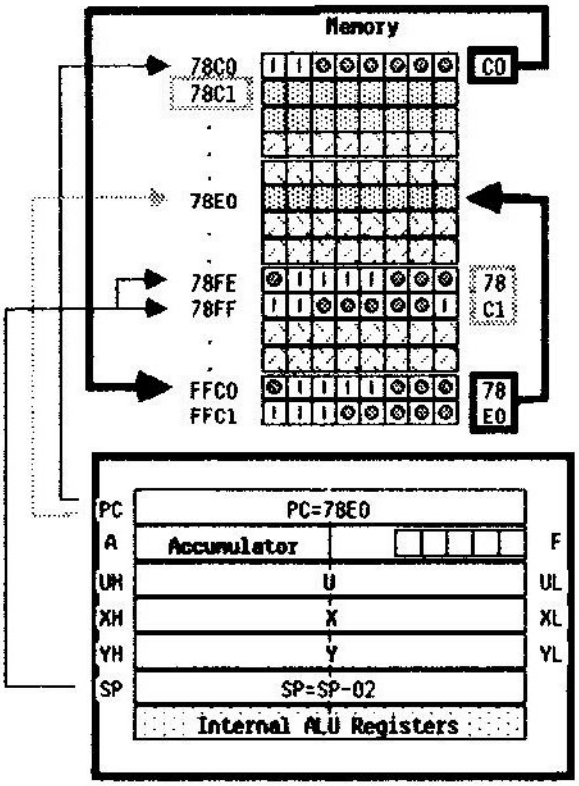
C0	C2	C4	C6	C8	CA	CC	CE
D0	D2	D4	D6	D8	DA	DC	DE
E0	E2	E4	E6	E8	EA	EC	EE
F0	F2	F4	F6	F8	FA	FC	FE

The special opcode relationship is this: the opcode itself serves as the reference to the address on page FF where the absolute two-byte address of the subroutine is stored!

Note that only the upper quadrant of the base page (FF) can be reached by these directives. Of course, just as in the case of the two-byte call, all the locations are "even-valued" as the odd-valued locations contain the second part of each two-byte address that is stored on the base page.

But, these directives all operate in the same

 Copyright 1983 POCKET COMPUTER NEWSLETTER



CPU

manner as the jump to subroutine directive. When executed, the address of the byte that follows the opcode will be stored on the stack as the "return to" address. It is a pretty clever little directive, one not found on typical CPUs. And, in the case of its application in the Sharp PC-1500, its use ends up saving a significant amount of memory.

For instance, working with the example cited earlier (20 calls to a 51-byte block of code), the use of this type of directive would reduce memory storage requirements to just 73 bytes. (Don't forget the two-byte address stored on the base page when considering this case!)

Alas, however, from a practical standpoint the one byte call directive will be of little use unless you can get your hands on a loose LH5801 or you plan to tap into the ROM coding already present in the PC. It is worth knowing about for just such an eventuality, however!

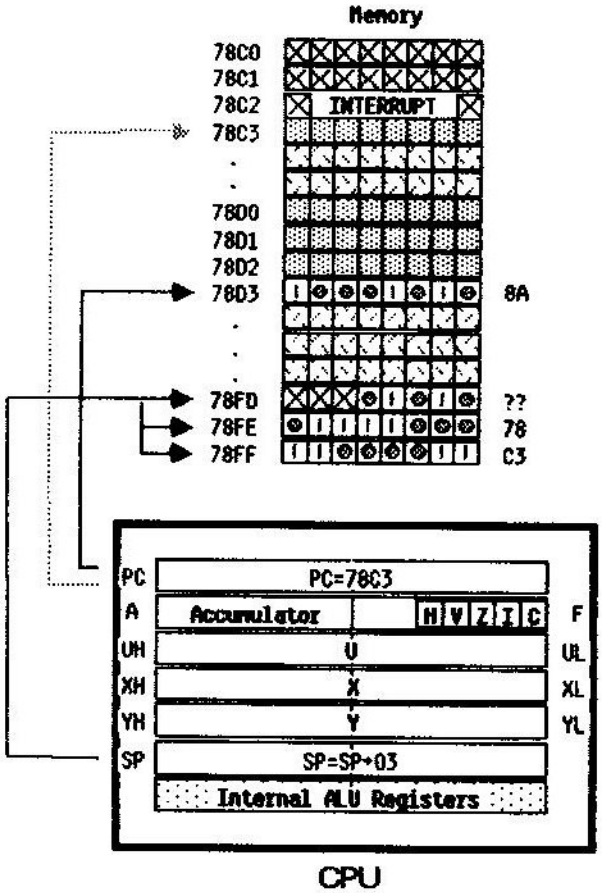
A Special Return Instruction

When certain types of external devices need to communicate with a computer they generate what is known as an *interrupt signal*. The LH5801 CPU responds to such a signal by automatically performing a special kind of jump to subroutine operation. That is, it stops whatever it is currently doing (at the conclusion of whatever instruction it is executing) and pushes the address of the next instruction it would normally execute onto the stack. It also places one more item of information in the stack: the contents of the CPU flag (F) register! It does this so that the present state of all the CPU (programmable) flags can be preserved. Finally, it jumps off to one of several addresses that are stored in the upper locations of page FF. Just which one is dependent upon the type of interrupt that is being performed. The details of this selection will be reserved for later discussion.

It then proceeds to execute whatever instructions it finds at the address it has been directed to as though it were a subroutine. That is, it keeps going until it finds a return instruction. However, since it stored three bytes of information on the stack, it needs to perform a special kind of return procedure in order to return to the place in memory from whence it was originally operating when the interrupt took place. There is a special return directive that performs this extraordinary procedure:

Mnemonic	Code
RTI	8A

The mnemonic stands for "return from an interrupt subroutine." When it is executed, the CPU will first increment the value in the stack pointer, then fetch whatever is in the memory location pointed to by the stack pointer and place it into the CPU flag register. *This action affects all of the CPU flags by restoring them to whatever state*



CPU

they had when the interrupt caused them to be stored! The CPU then increments the value in the stack pointer again, fetches the high part of the return address from the stack, increments the stack pointer once more and fetches the low part of the return address. The return address obtained is placed into the program counter, thereby causing the CPU to resume activity with the next instruction in memory after the one being performed when the interrupt originally occurred.

In summary, the RTI instruction is just like a RTS except that it takes one more byte off the stack. This extra byte is used to set the states of the various CPU flags. Thus, while a regular return directive never alters CPU flags, all flags will be affected by a return from interrupt command.

Since the use of interrupts is strictly limited to special situations dealing with external hardware, many programmers will never have to be concerned with its practical application. Just make sure you don't use it accidently, however. Else you could be in for all kinds of surprises caused by a misaligned stack pointer and a revised set of flag conditions!

Decimal Addition

One of the first instructions I introduced in the first part of this series was the regular old binary addition directive. The LH5801 CPU is capable of performing another type of addition directive known as decimal adjusted addition. It is somewhat more complicated than an ordinary binary add. Hence an explanation of its operation was not included in the previous discussion. After all, I didn't want to scare you off when you were just getting started. If you are still with me at this point, I figure you have enough gumption to deal with just about anything dished out!

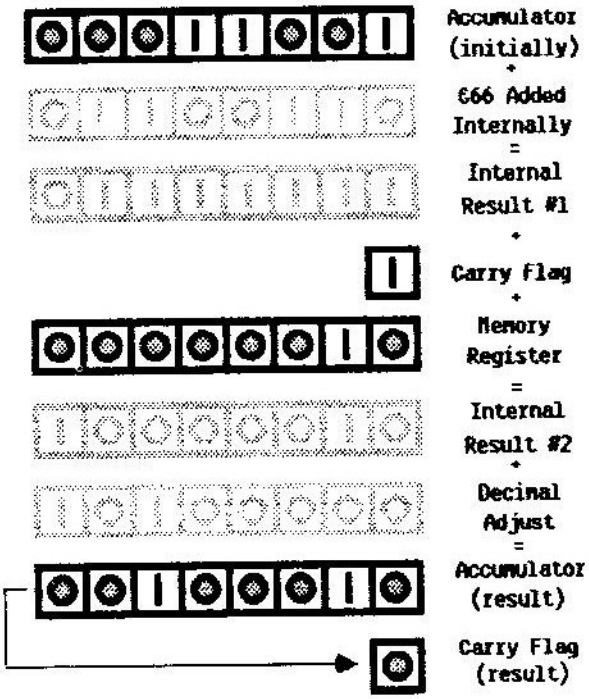
Basically the way a decimally adjusted operation works is to divide a byte into two parts consisting of four bits each. (Four bits are sometimes referred to as a nibble in data processing lingo.) Each 4-bit half-byte is then treated as a decimal value. That is, it is "adjusted" (during the operation of a "decimal" instruction) so that the four bits represent values in the range 0 through 9. Any carry from this procedure is either reflected in the H (half-carry) flag or C (carry) flag depending on which nibble is involved.

The essence of all this is that you can effectively work with numbers in decimal format (actually, binary coded decimal [BCD] format) instead of always having to work in strict binary notation. It turns out that there are drawbacks to working in this format so that it is not as easy to use as it sometimes appears at first glance. However, some programmers like to work with BCD procedures. The LH5801 has the fundamental decimal-oriented instructions that lets those so-inclined work in this mode. Here, for starters, are the three decimal add directives that allow the contents of the memory location pointed to by the U, X or Y registers to be decimal added to the accumulator:

Mnemonic	Code
DADA (U)	AC
DADA (X)	8C
DADA (Y)	9C

And here is the formal explanation of how they operate: (1) The hexadecimal value 66 is added (internally) to the initial contents in the accumulator (which is assumed to be in BCD form), (2) The initial content of the carry flag (C) is added to the least significant decimal digit position, (3) The value in the location pointed to by the data pointer register (which is assumed to be in BCD form) is added to the accumulator, (4) The intermediate result obtained at this point is adjusted to yield final BCD digits by adding in a value that is a function of the half-carry (H) flag and carry (C) flags, (5) Any BCD overflow from the final result in the most significant BCD digit position will be reflected in the carry (C) flag, (6) The status of the Z and V flags may also be affected by the results of the operation, (7) The original contents of the memory location (addend) are not altered by the procedure.

The decimal adjusting operation performed in step (4) of the above description consist of adding particular values (internally) to the accumulator depending on the states of the H and C flags. *It is essential to realize that these flags are operating in such a manner so as to indicate overflows of*



 Copyright 1983 POCKET COMPUTER NEWSLETTER

BCD values from each nibble during this process. The exact value that will be added to "adjust" the results may be determined from this table:

Carry	Half-Carry	Decimal-Adjust Value
0	0	9A
0	1	A0
1	0	FA
1	1	00

The accompanying diagram representing the operation of a decimal addition instruction should help clarify what often seems to be a somewhat obscure procedure. Whether the actual internal process is understood by the programmer is not really critical. The important concept is to realize that if you start out with values in BCD format, you will end up with values in BCD format, provided you utilize these special BCD directives.

Decimal Subtraction

A similar set of directives exist to facilitate subtraction in BCD format:

Mnemonic	Code
DSBA (U)	2C
DSBA (X)	0C
DSBA (Y)	1C

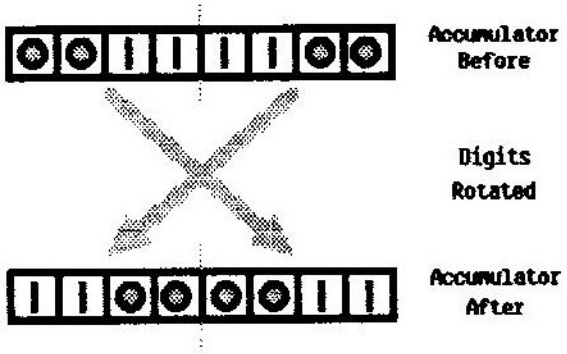
The rules applying to the operation of these directives are as follows: (1) Decimal adjusted arithmetic is performed, (2) The status of the carry flag serves as a borrow indicator at the start of the operation, (3) The value in the accumulator is assumed to be in BCD form at the beginning of the operation, (4) The content of the memory location pointed to by the data pointer register (assumed to be in BCD form) is deducted from the accumulator, (5) The result is decimal adjusted using the status of the half-carry (H) flag and carry (C) flags to determine the adjusting quantity, (6) Any BCD underflow from the most significant BCD digit will be reflected by the status of the carry (C) flag at the conclusion of the operation, (7) The status of the Z and V flags may also be affected by the results of the operation, (8) The original contents of the memory (subtrahend) location are not altered by the procedure.

The adjusting value used in step (4) above, which is dependent upon the values of the carry and half-carry flags at the time of the operation, is the same as that given in the table presented for the decimal addition directives.

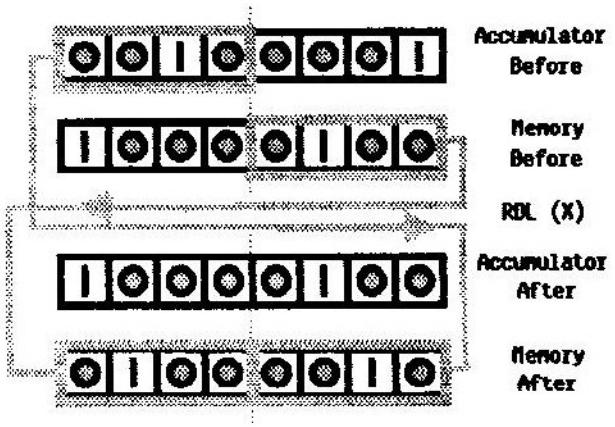
BCD Rotate Instructions

To facilitate working in BCD notation, the LH5801 can also perform several kinds of BCD (or nibble) rotate and swap operations. This permits quick and easy manipulation of BCD digits within the accumulator or between the accumulator and a location in memory.

The simplest of these directives is the RDA (rotate digits in the accumulator) command. This effectively swaps the right and left BCD digits (nibbles) within the A register.



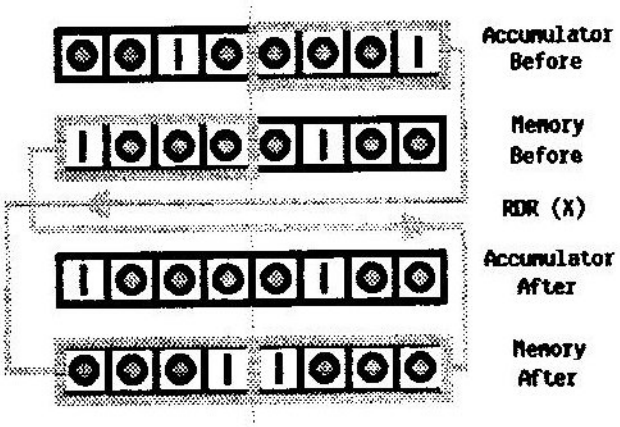
There is also a command that rotates the least significant nibble (BCD digit) of the accumulator into the most significant nibble of the memory location pointed to by the contents of the X register. As this is done, the original most significant nibble of the memory location is shifted to the right, to occupy the least significant nibble position. This BCD digit is also copied into the most significant nibble of the accumulator. Finally, the original least significant nibble of the memory location is transferred to the right-most nibble of the accumulator. The result of all this is called a *digit rotate right* and has the mnemonic RDR (X). The accompanying diagram will help clarify what can be a somewhat confusing operation.



In summary, what effectively happens is that the accumulator ends up containing the original contents of the memory location, while the memory location ends up having the former least significant nibble of the accumulator in its most significant bits and the former most significant nibble of itself shifted over to its least significant bits. *You should note that the original most significant BCD in the accumulator will be lost by the execution of this directive!*

A similar directive with the mnemonic RDL (X)

is used to rotate digits between the memory location pointed to by the address in the X register and the accumulator, in the opposite direction. The most significant BCD digit of the accumulator is passed to the least significant nibble of the memory location. The original most significant BCD digit of the memory location is transferred to the same position in the accumulator. The original least significant nibble of the memory location is shifted left to the most significant nibble position. It is also passed to the least significant BCD digit position in the accumulator. This is the so-called *digit rotate right* command. Again, the diagram can help illustrate the operation.



In summary, the accumulator ends up having the original contents of the memory location. The memory location ends up with its lowest nibble shifted over to the high nibble position. The original high nibble of the accumulator is transferred to the low nibble of the memory register. *The original least significant BCD digit in the accumulator is lost as a consequence of performing this instruction.*

The mnemonics and machine codes for these digit rotate directives are summarized here:

Mnemonic	Code
RDA	F1
RDL (X)	D7
RDR (X)	D3

There is one more important point to remember about the digit rotate instructions: none of the CPU flags are affected by their execution.

A Good For Nothing Instruction

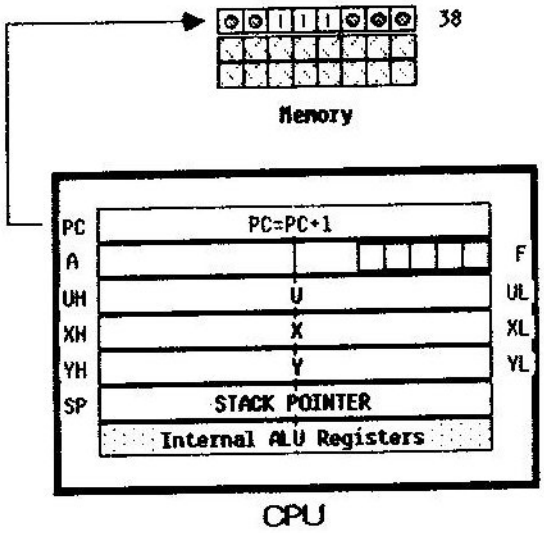
Would you believe there is a instruction that does absolutely nothing? There is. Whenever the CPU sees this directive, it just shrugs it chips, advances the program counter and goes on to the next location in memory.

But, is this "do nothing" instruction really completely useless? No. Most machine language programmers love having it available. It is often used as a "filler" at locations in memory where one suspects programming changes may have to be made. And, it is often used to "patch over" sections of code that are no longer desired in a program.

The instruction is referred to as a *no-operation* (fondly shortened to "no-op") by those in the trade. Its mnemonic in this text is aptly:

Mnemonic	Code
NOP	38

Since the instruction does nothing but allow the program counter to advance to the next memory address, you know that it does not affect the status of the CPU flags.



You will learn just how handy this directive can be the first time you discover you need to replace a three-byte directive with a two-byte one in the middle of a block of machine code!

Switching ROMs or Whatever

The Sharp PC-1500 and Radio Shack PC-2 units can, as you well know, be connected to a variety of peripheral devices, such as a printer or RS-232C communications interface. These devices typically contain ROM(s) (in the address range 8000 - BFFF). Some of them contain several ROMs in this same address range.

You know that the LH5801 CPU cannot possibly execute two different sets of instructions stored in two different ROMs at the same time. In fact, if two ROMs residing in the same memory addresses were activated simultaneously, the result would be electronic chaos. No, the fact of the matter is that the computer must only allow one ROM (within a given address range) to be active at a time. That is, it must tell one ROM when to be active and in so doing tell any competing ROM to cut itself out of the circuit.

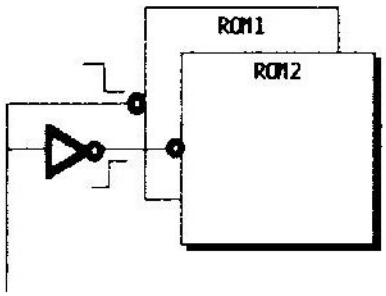
This is a rather straightforward procedure from

 Copyright 1983 POCKET COMPUTER NEWSLETTER

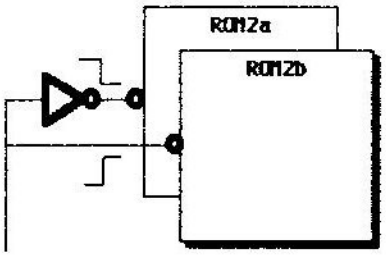
an electronic circuit standpoint. All it takes is a signal that can go from a high state to a low state or the reverse. The output from a settable flip-flop circuit is just what is needed for this chore. Guess what? The LH5801 has several flip-flops (besides the flags that you already know about) that can be controlled by special directives.

Since the PC-1500 uses these flip-flops to control which ROM is activated in an external device, they have been assigned mnemonics that relate to this common use. However it will be pointed out that when the PC is not connected to a device that uses these signals, they could be used for whatever purposes a programmer desired!

One signal line comes out to pin 15 (on the 60 pin connector) of the PC. We will refer to this signal line as the ROM1/ROM2 line. When the logic state of this line is at the low level, it is used to activate ROM1 in an external device. When it is in the high condition, it activates ROM2. You can set this line to the state that activates ROM1 (low level) by executing the ROM1 instruction. Or, you can set it to activate ROM2 (high level) by using the ROM2 directive. Neat, eh?



When 2 ROMs occupy same address range in memory, the ROM1 and ROM2 commands can activate one at a time.



The ROM2a and ROM2b commands can similarly be used to select between another two sets of ROMs.

Similarly another signal line we will refer to as ROM2a/ROM2b comes out to pin 16 of the PC connector. When the signal level on this pin is high it can be used to activate ROM2a. When it is low it switches the external device over to ROM2b. You guessed it: executing the instruction ROM2a puts this line in the high state, while using ROM2b will place it in the low condition. (The RS-232 interface uses this signal to switch between two ROMs that share the address range 8000 - 9FFF.)

Mnemonic	Code
ROM1	B8
ROM2	A8
ROM2a	E1
ROM2b	E3

You will want to recall these instructions the next time you start thinking about how you might control a homebrew external device of your own design. They would seem to provide interesting possibilities in many types of applications.

Lookup Tables

Believe it or not, you have now learned the great majority of all the instructions that can be performed by the LH5801 CPU. There are some more -- those that make up what is referred to as the extended instruction set -- but many of these are simply extensions of those we have already covered. In any event, it is appropriate to spend a little time at this point, getting into the art and science of machine language programming at the practical, hands-on, level.

The first order of business is to understand that it is not necessary to memorize every mnemonic for every instruction that the machine can perform. While some programmers eventually do this, especially after they have worked with a particular CPU for a long time, I do not recommend that anyone put a lot of effort into acquiring such a rote skill. The important thing is to know what classes of instructions -- loads, stores, adds, subtracts, logical operations, etc. -- are available on the machine and what types of addressing modes may be used. This is the conceptual level at which the successful programmer must learn to think.

All of the other aspects of machine language programming may be accomplished by rote table lookup! All you need to have on hand is the appropriate lookup tables. Two such tables are provided on adjoining pages.

The first table is organized according to machine code values. It is used when you want to disassemble machine code. That is, when you need to know what a particular numeric value means to the CPU. Alongside each value that can serve as a CPU opcode is its mnemonic representation. You should note that some numeric values do not invoke a *defined* response from the CPU. *These undefined values should never be used as opcodes within a program.* In other words, the fact that a numeric value is not defined as representing an instruction does not mean that it can safely be used as a NOP (no-operation) directive. It should always be assumed that the use of such a value could produce *unpredictable* results. Hence, they should *never* serve as opcodes within a program.

The second table is organized according to mnemonic representation. It is used when you want to assemble a program. That is, when you want to convert a list of mnemonics (with which the

Lookup Table In Machine Code Order

Code	Mnemonic
00	SUBA XL
01	SUBA (X)
02	ADDA XL
03	ADDA (X)
04	LDA XL
05	LDA (X)
06	CPA XL
07	CPA (X)
08	STA XH
09	ANDA (X)
0A	STA XL
0B	ORA (X)
0C	DSBA (X)
0D	EORA (X)
0E	STA (X)
0F	BITA (X)
10	SUBA YL
11	SUBA (Y)
12	ADDA YL
13	ADDA (Y)
14	LDA YL
15	LDA (Y)
16	CPA YL
17	CPA (Y)
18	STA YH
19	ANDA (Y)
1A	STA YL
1B	ORA (Y)
1C	DSBA (Y)
1D	EORA (Y)
1E	STA (Y)
1F	BITA (Y)
20	SUBA UL
21	SUBA (U)
22	ADDA UL
23	ADDA (U)
24	LDA UL
25	LDA (U)
26	CPA UL
27	CPA (U)
28	STA UH
29	ANDA (U)
2A	STA UL
2B	ORA (U)
2C	DSBA (U)
2D	EORA (U)
2E	STA (U)
2F	BITA (U)
30	
31	
32	
33	
34	
35	
36	
37	
38	NOP
39	
3A	
3B	
3C	

Code	Mnemonic
3D	
3E	
3F	
40	INXL
41	STAI (X)
42	DEXL
43	STAD (X)
44	INX
45	LDAI (X)
46	DEX
47	LDAD (X)
48	LDXH #nn
49	AND (X) #nn
4A	LDXL #nn
4B	OR (X) #nn
4C	CPXH #nn
4D	BIT (X) #nn
4E	CPXL #nn
4F	ADNC (X) #nn
50	INYL
51	STAI (Y)
52	DEYL
53	STAD (Y)
54	INY
55	LDAI (Y)
56	DEY
57	LDAD (Y)
58	LDYH #nn
59	AND (Y) #nn
5A	LDYL #nn
5B	OR (Y) #nn
5C	CPYH #nn
5D	BIT (Y) #nn
5E	CPYL #nn
5F	ADNC (Y) #nn
60	INUL
61	STAI (U)
62	DEUL
63	STAD (U)
64	INU
65	LDAI (U)
66	DEU
67	LDAD (U)
68	LDUH #nn
69	AND (U) #nn
6A	LDUL #nn
6B	OR (U) #nn
6C	CPUH #nn
6D	BIT (U) #nn
6E	CPUL #nn
6F	ADNC (U) #nn
70	
71	
72	
73	
74	
75	
76	
77	
78	
79	

Code	Mnemonic
7A	
7B	
7C	
7D	
7E	
7F	
80	SUBA XH
81	FBNC #nn
82	ADDA XH
83	FBC #nn
84	LDA XH
85	FBNH #nn
86	CPA XH
87	FBH #nn
88	BNZD #nn
89	FBNZ #nn
8A	RTI
8B	FBZ #nn
8C	DADA (X)
8D	FBNV #nn
8E	FB #nn
8F	FBV #nn
90	SUBA YH
91	RBNC #nn
92	ADDA YH
93	RBC #nn
94	LDA YH
95	RBNH #nn
96	CPA YH
97	RBH #nn
98	
99	RBNZ #nn
9A	RTS
9B	RBZ #nn
9C	DADA (Y)
9D	RBNV #nn
9E	RB #nn
9F	RBV #nn
A0	SUBA UH
A1	SUBA nnnn
A2	ADDA UH
A3	ADDA nnnn
A4	LDA UH
A5	LDA nnnn
A6	CPA UH
A7	CPA nnnn
A8	ROM2
A9	ANDA nnnn
AA	LDS# nnnn
AB	ORA nnnn
AC	DADA (U)
AD	EORA nnnn
AE	STA nnnn
AF	BITA nnnn
B0	
B1	SUBA #nn
B2	
B3	ADDA #nn
B4	
B5	LDA #nn
B6	

Code	Mnemonic
B7	CPA #nn
B8	ROM1
B9	ANDA #nn
BA	JMP nnnn
BB	ORA #nn
BC	
BD	EORA #nn
BE	JSR nnnn
BF	BITA #nn
C0	CALL C0
C1	CANC #nn
C2	CALL C2
C3	CAC #nn
C4	CALL C4
C5	CANH #nn
C6	CALL C6
C7	CAH #nn
C8	CALL C8
C9	CANZ #nn
CA	CALL CA
CB	CAZ #nn
CC	CALL CC
CD	CALL #nn
CE	CALL CE
CF	CAV #nn
D0	CALL D0
D1	RRCA
D2	CALL D2
D3	RDR (X)
D4	CALL D4
D5	SRA
D6	CALL D6
D7	RDL (X)
D8	CALL D8
D9	SLA
DA	CALL DA
DB	RLCA
DC	CALL DC
DD	INA
DE	CALL DE
DF	DEA
E0	CALL E0
E1	ROM2a
E2	CALL E2
E3	ROM2b
E4	CALL E4
E5	
E6	CALL E6
E7	
E8	CALL E8
E9	AND nnnn #nn
EA	CALL EA
EB	OR nnnn #nn
EC	CALL EC
ED	BIT nnnn #nn
EE	CALL EE
EF	ADNC nnnn #nn
F0	CALL F0
F1	RDA
F2	CALL F2
F3	

Code	Mnemonic
F4	CALL F4
F5	STI (X)(Y)
F6	CALL F6
F7	CPAI (X)
F8	CALL F8
F9	CLRC
FA	CALL FA
FB	SETC
FC	CALL FC
FD	
FE	CALL FE
FF	

 Copyright 1983 POCKET COMPUTER NEWSLETTER

Lookup Table in Mnemonic Order

Mnemonic	Code
	30
	31
	32
	33
	34
	35
	36
	37
	39
	3A
	3B
	3C
	3D
	3E
	3F
	70
	71
	72
	73
	74
	75
	76
	77
	78
	79
	7A
	7B
	7C
	7D
	7E
	7F
	98
	B0
	B2
	B4
	B6
	BC
	E5
	E7
	F3
	FD
	FF
ADDA #nn	B3
ADDA (U)	23
ADDA (X)	03
ADDA (Y)	13
ADDA nnnn	A3
ADDA UH	A2
ADDA UL	22
ADDA XH	82
ADDA XL	02
ADDA YH	92
ADDA YL	12
ADNC (U) #nn	6F
ADNC (X) #nn	4F
ADNC (Y) #nn	5F
ADNC nnnn #nn	EF
AND (U) #nn	69
AND (X) #nn	49
AND (Y) #nn	59
AND nnnn #nn	E9

Mnemonic	Code
ANDA #nn	B9
ANDA (U)	29
ANDA (X)	09
ANDA (Y)	19
ANDA nnnn	A9
BIT (U) #nn	6D
BIT (X) #nn	4D
BIT (Y) #nn	5D
BIT nnnn #nn	ED
BITA #nn	BF
BITA (U)	2F
BITA (X)	0F
BITA (Y)	1F
BITA nnnn	AF
BNZD #nn	88
CAC #nn	C3
CAH #nn	C7
CALL #nn	CD
CALL C0	C0
CALL C2	C2
CALL C4	C4
CALL C6	C6
CALL C8	C8
CALL CA	CA
CALL CC	CC
CALL CE	CE
CALL D0	D0
CALL D2	D2
CALL D4	D4
CALL D6	D6
CALL D8	D8
CALL DA	DA
CALL DC	DC
CALL DE	DE
CALL E0	E0
CALL E2	E2
CALL E4	E4
CALL E6	E6
CALL E8	E8
CALL EA	EA
CALL EC	EC
CALL EE	EE
CALL F0	F0
CALL F2	F2
CALL F4	F4
CALL F6	F6
CALL F8	F8
CALL FA	FA
CALL FC	FC
CALL FE	FE
CANC #nn	C1
CANH #nn	C5
CANZ #nn	C9
CAV #nn	CF
CAZ #nn	CB
CLRC	F9
CPA #nn	B7
CPA (U)	27
CPA (X)	07
CPA (Y)	17
CPA nnnn	A7

Mnemonic	Code
CPA UH	A6
CPA UL	26
CPA XH	86
CPA XL	06
CPA YH	96
CPA YL	16
CPAI (X)	F7
CPUH #nn	6C
CPUL #nn	6E
CPXH #nn	4C
CPXL #nn	4E
CPYH #nn	5C
CPYL #nn	5E
DADA (U)	AC
DADA (X)	8C
DADA (Y)	9C
DEA	DF
DEU	66
DEUL	62
DEX	46
DEXL	42
DEY	56
DEYL	52
DSBA (U)	2C
DSBA (X)	0C
DSBA (Y)	1C
EORA #nn	B0
EORA (U)	20
EORA (X)	00
EORA (Y)	10
EORA nnnn	A0
FB #nn	8E
FBC #nn	83
FBH #nn	87
FBNC #nn	81
FBNH #nn	85
FBNV #nn	80
FBNZ #nn	89
FBV #nn	8F
FBZ #nn	8B
INA	D0
INU	64
INUL	60
INX	44
INXL	40
INY	54
INYL	50
JMP nnnn	BA
JSR nnnn	BE
LDA #nn	B5
LDA (U)	25
LDA (X)	05
LDA (Y)	15
LDA nnnn	A5
LDA UH	A4
LDA UL	24
LDA XH	84
LDA XL	04
LDA YH	94
LDA YL	14
LDAD (U)	67

Mnemonic	Code
LDAD (X)	47
LDAD (Y)	57
LDAI (U)	65
LDAI (X)	45
LDAI (Y)	55
LDS# nnnn	AA
LDUH #nn	68
LDUL #nn	6A
LDXH #nn	48
LDXL #nn	4A
LDYH #nn	58
LDYL #nn	5A
NOP	38
OR (U) #nn	6B
OR (X) #nn	4B
OR (Y) #nn	5B
OR nnnn #nn	EB
ORA #nn	B8
ORA (U)	28
ORA (X)	08
ORA (Y)	18
ORA nnnn	A8
RB #nn	9E
RBC #nn	93
RBH #nn	97
RBNC #nn	91
RBNH #nn	95
RBNV #nn	90
RBNZ #nn	99
RBV #nn	9F
RBZ #nn	9B
RDA	F1
RDL (X)	D7
RDR (X)	D3
RLCA	DB
ROM1	B8
ROM2	A8
ROM2a	E1
ROM2b	E3
RRCA	D1
RTI	8A
RTS	9A
SETC	FB
SLA	D9
SRA	D5
STA (U)	2E
STA (X)	0E
STA (Y)	1E
STA nnnn	AE
STA UH	28
STA UL	2A
STA XH	08
STA XL	0A
STA YH	18
STA YL	1A
STAD (U)	63
STAD (X)	43
STAD (Y)	53
STAI (U)	61
STAI (X)	41
STAI (Y)	51

Mnemonic	Code
STI (X)(Y)	F5
SUBA #nn	B1
SUBA (U)	21
SUBA (X)	01
SUBA (Y)	11
SUBA nnnn	A1
SUBA UH	A0
SUBA UL	20
SUBA XH	80
SUBA XL	00
SUBA YH	90
SUBA YL	10

machine language programmer works and thinks) into the machine codes (numeric values, binary patterns) that the CPU utilizes.

These two tables contain all the instructions that have been presented so far in this text. They will be most valuable to you in the future. Treasure them well!

Initial Program Development

No machine language programmers worth their salt barge into writing a machine language routine. Those that attempt such a brute-force usually end up wasting a lot of time. The number one rule in this kind of programming is: *decide exactly what it is that the computer is to do!* The second rule is: *write it down.*

Now this may seem silly to have to spell out. But the fact of the matter is that many people who learn programming using a high language level such as BASIC fail to realize how rigorous the process becomes at the machine language level. If you make a mistake or change your mind when using BASIC, you can usually just insert a new line here and delete a line or two there to correct the problem. Insertions and deletions don't go over so easily when using machine language.

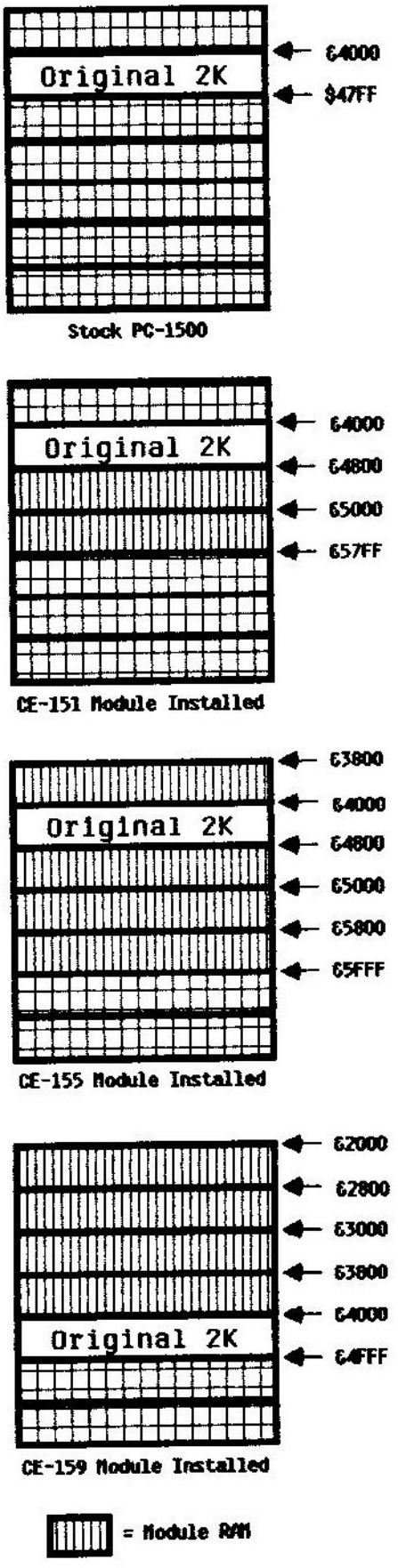
The act of actually writing down, using simple English, the proposed operation of a machine language routine serves several purposes. For one, the process forces a careful review of what you have been planning. This methodical outlining of a process frequently reveals flaws that may have been invisible when the matter was mulled over purely in the slippery recesses of your mind.

Second, this written record can become a guide and checklist as the actual machine language routine is developed. In view of the fact that it may take many hours -- spread over a period of days -- to actually code a substantial routine, having such a written plan to refer to can save a lot of time. It is amazing how much your mind can forget if it is not periodically refreshed. Proper work habits can have a powerful impact on the overall progress one achieves while doing this type of programming.

Of course, if you want to amplify your written descriptions by creating flowcharts, then you should certainly do so. As a general rule, because of the level of complexity involved, the more programming aids you can provide for yourself, the smoother things will go at the machine language level.

Mapping Memory

Another parameter to think about before you start writing a program is where you will store the instructions in memory. You will also need to consider whether there will be any locations used for data storage. It is a good idea to at least rough



 Copyright 1983 POCKET COMPUTER NEWSLETTER

out what is referred to as a *memory map*. This is simply a block diagram that indicates what ranges of memory addresses will be used for specific or general purposes.

The actual locations you use in memory to store a program and/or data will, of course, depend on how much memory you have installed in your PC.

If, for instance, you have a PC-1500 with a CE-155 RAM module installed, then your RAM addresses will run from &3800 - &5FFF. This gives you 10K of RAM. Other modules give other address ranges and amounts of memory as illustrated in the accompanying diagrams.

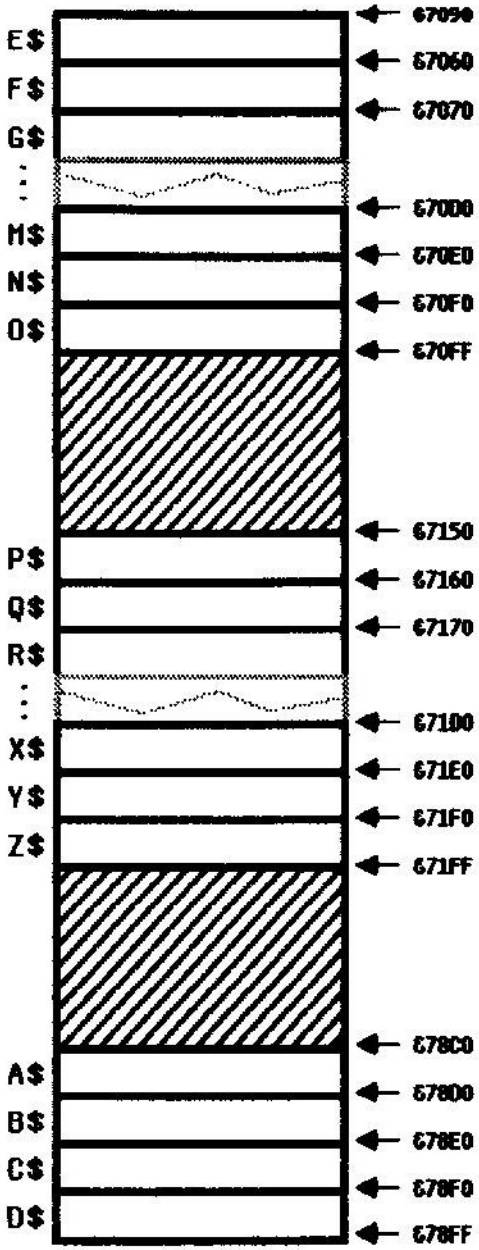
Remember that normally the BASIC interpreter utilizes all of available RAM. However, you can protect (block off) a section of RAM so that BASIC does not use it. This is accomplished by giving the NEW XXXX directive where XXXX represents an address. When invoked, BASIC will not store a user's program below the specified address. Thus, if you have an 8K (CE-155) module installed and you enter NEW &4000, then the area below $4000 (&3800 - &3FFF) will be protected from use by a BASIC program.

If you only plan to deal with small machine language routines, you may want to tuck them away in the special RAM locations normally used to store BASIC variables. Accompanying diagrams identify the address ranges used for such storage. However, if you do use these areas for machine language instructions, it is absolutely vital that you remember the following: *never use any variable whose normal storage location has been usurped for machine language purposes and make sure that you do not use the BASIC statement CLEAR.* If you fail to heed this advice you will find your ML routines suddenly peppered with zero bytes or other unwanted values.

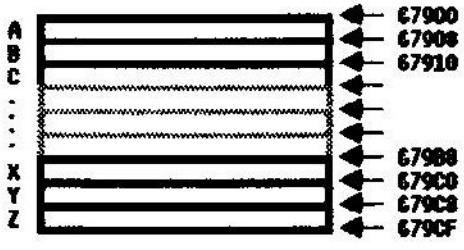
A Machine Language Worksheet

Once you have decided where you plan to store a routine and any associated data and you know exactly what the program is going to do, you can start thinking about what machine language directives to use. As you select each instruction, it is a good idea to write down the mnemonic for the instruction opcode along with any operands or addressing information on a program worksheet. You may also want to assign labels as substitutes for addresses at this point. And, it is a good idea to write down a description of exactly what the instruction will accomplish in relation to the overall program.

A copy of the type of worksheet I like to use for machine language programming is provided for you to make duplicates of, if desired. Development work is done using the right side of the worksheet (in the three columns marked Labels, Mnemonics and Comments). Later, the mnemonics can be "assembled" into machine code and assigned to specific memory locations by filling in columns on the left side of the sheet.



BASIC string variables memory map.



BASIC numeric variables memory map.

How to Assemble a Program

The process of converting the mnemonics used by a programmer into the actual numeric values used by a computer when the program is executed is known as *assembling a program.* A program in mnemonic form is often referred to as the *source code.* The final product of the assembly process is called the *object code.*

The converting of source code (the mnemonics) to object code (the machine code) is a relatively simple process. However, the process can be rather tedious. This is especially true if the program is lengthy or if it is frequently revised. For that reason, most professional machine language programmers like to work with an *assembler program.* This is a program that automatically processes a text file containing mnemonics and translates it to machine code.

Naturally, an assembler program must be especially designed to assemble code for a specific CPU. Since the process involves translating mnemonics into machine codes (table lookup), one of the things necessary for the functioning of such a program is a complete lookup table. Such a table takes a fair amount of memory when stored in the memory of a computer. I estimate that a fully functional assembler program for the PC-1500 might consume some 7 to 8 kilobytes or more of memory. Ten-kilobyte PC-1500 systems are common these days (since the basic 2K PC can be expanded by adding a CE-155 8K RAM expansion module). Even 18-kilobyte systems can now easily be configured (by installing a CE-161 16K RAM module). However, there are still substantial reasons for not attempting to work with a true assembler program in a PC-1500.

One of these reasons is that, when an assembler program is used, it is necessary to pass the source code (mnemonic listing) to the assembler. The ideal way to do this is to store the mnemonic listing as a text file in memory. Of course, to do this you now need storage room for that text file, to say nothing of room for an editor program to create such a file. Finally there is the problem of what to do with the object code created by the assembler program. Again, a nice solution is to be able to store the finished code directly in memory. Alas, this eats up still more of that precious commodity: RAM.

Conventional desktop computer systems often get around the problem of handling source and object code by using what is referred to as a *multiple-pass* system. The source listing is first created using an editor and stored on an external floppy disk. Then the assembler program is loaded into memory. It processes the mnemonic source file from the disk system and stores the object code it produces back on the disk. This final machine code can then be loaded into the appropriate memory addresses and executed as a machine language program. Since a floppy disk can process files rapidly, the entire process of assembling even a relatively large program might take just a few minutes on a desktop system.

Alas, following the same type of procedure using the audio tape storage capabilities of the PC-1500 would consume vast amounts of time. It could take 30 minutes or more just to attempt the assembly of even a small program. This hardly seems practical. Even a beginner can assemble a modest machine language routine using manual table lookup methods in substantially less time. Thus, the manual method will be used in this text!

The process is easy, especially if you write down the mnemonics for the instructions you want to use on an appropriately formatted *worksheet.*

All you have to do is decide the memory address of where you want to start storing the object code. Note this address in the PG (high address byte value) and LC (low address byte value) columns on the worksheet. Now look up the machine code for the mnemonic that is being translated (using the alphabetically-arranged lookup table). Place this code value in the column identified as B1 on the worksheet opposite the corresponding mnemonic. Next, determine whether the instruction uses additional bytes. That is, if it must be followed by addressing information or immediate data. (This information is also available from the lookup table.) If so, insert the necessary information (in numeric form), a byte at a time, in the columns titled B2 through B5.

Once this has been accomplished for an instruction, you can go on to the next line on the worksheet. At this point you can fill in the starting address of the next instruction simply by counting the number of bytes utilized by the preceding directive! Thus, if a three-byte directive was stored beginning at address &7151, then the next instruction would be stored beginning at address &7154.

The only time things can get a little difficult is when the program contains what is known as *forward references.* That is, when it contains jumps, calls or branches to sections of the program that have not yet been assembled. The reason this can cause problems is because you will not initially know the absolute addresses at which such referred-to directives will be stored. Yet, to properly assemble the machine code for such references, you will eventually need to specify this information precisely.

Ah, but the situation is not hopeless. Anytime you encounter such a directive, you simply set aside the appropriate number of bytes needed to

PC	LC	B1	B2	B3	B4	B5	LABELS	MNEMONICS	COMMENTS

DATE:_____ PROGRAM:_____ ROUTINE:_____ PAGE ____ OF ____

Copyright 1983 POCKET COMPUTER NEWSLETTER

complete the reference. In the case of branches, this is always one byte. A jump directive requires two extra bytes to specify an absolute address. A call directive will also usually require two bytes. (You may remember that several special classes of call directives may use just one or no additional bytes. However, since the vectors used by these types of calls are occupied by the BASIC ROM in the PC-1500/PC-2, most programmers will not have occassion to use these special cases.) By set aside, I mean mark the appropriate columns (such as B2 and/or B3) on the worksheet as being reserved. You then count those bytes in order to determine the starting address of the next instruction to be noted on your worksheet. When the actual address of the referred-to instruction is eventually determined, you can go back and fill in the appropriate values in the reserved column locations. A few programming examples in the future will clarify this procedure. It is quite straightforward, though it does require taking care to insure accuracy.

This matter of forward references within a machine language program is the reason programmers like to work with *labels* instead of absolute addresses. The column titled "Labels" on the worksheet is provided for this purpose. Whenever you want to refer to particular points within a program, you can tag them by creating your own reference names. You can then specify references to those points using the label names. Thus, instead of the mnemonic for a jump directive being JMP &7060, you might write JMP THERE (in the worksheet column titled "Mnemonics"). The referred-to point in the program would then be appropriately identified on the worksheet by placing the name of that tag (THERE) in the "Label" column. Again, this methodology will be illustrated in future programming examples.

Loading and Testing a ML Program

Once the machine code for a program has been assembled by hand, it is necessary to load it into the appropriate locations in the memory of the PC.

In the case of a small routine, this can easily be accomplished using the POKE directive that is available in BASIC. The format of this statement is: POKE X,Y where X represent an address and Y stands for the code to be stored therein. There is an alternate format for this statement. The address specification may be followed by a series of data values separated by commas. Thus, a statement such as POKE &7050,&48,&71,&4A,&50 would specify that the hexadecimal values 48, 71, 4A and 50 were to be stored in the four bytes of memory starting at the hexadecimal address 7050.

If you plan on doing a substantial amount of ML programming, then the use of a so-called monitor program can be of great practical value. A monitor program in this context is one that assists the programmer in the process of storing machine codes into memory, examining and altering the contents of memory locations, and so forth. I would recommend that a serious ML enthusiast consider obtaining the Loader/Monitor/Disassembler that is sold by *PCM*. This is an integrated package of routines that greatly facilitates working in machine language on a PC-1500 or PC-2.

Once a ML routine has been stored in memory it may be executed by calling it from a BASIC program. Alternately, a monitor program may be used to check its operation. If a complex routine does not operate as expected, it may be necessary to debug the routine. Debugging ML routines can be very difficult without the assistance of a monitor program. Thus, it is extremely important to plan the operation of a ML routine carefully so as to guard against errors.

An errant ML program will often result in the PC "locking up." The only way to recover from such a state is through the use of the reset button on the back of the unit. As you undoubtably know, using the reset button can cause all of memory to be erased. You then have to start program loading all over. This can be a rather discouraging event. Careful planning and coding of your routines can significantly reduce the chances of this occurring. However, you probably won't consider yourself properly initiated into the realm of MLP until you have had a routine "bomb" and lockup your PC.

The only thing worse than lockup is to discover that you left out an important step in your program somewhere near the beginning. This means all of the instructions beyond that point must be relocated, forward references altered, and so forth. After a few mistakes of this nature, you will gain a profound appreciation for the cultivation of good programming work habits that can help reduce the chances for making such errors.

In the next section of this series I will present a number of practical ML programming routines. In addition to their educational value, these routines can be the start of your own personalized library of frequently used ML procedures and algorithms.

In the meantime, why not make some duplicates of the worksheet form and try your hand at creating some useful ML routines?

Machine Language Programming
the Sharp PC-1500 and Radio Shack PC-2
is published by
POCKET COMPUTER NEWSLETTER
P.O. Box 232, Seymour, CT 06483

FOR PC-1500 & PC-2 USERS

MACHINE LANGUAGE PROGRAMMING

Readers of the series *Machine Language Programming the Sharp PC-1500 and Radio Shack PC-2* (produced in 1983 as a separate publication by *PCW*) will especially appreciate the following article. As such readers know, the series was abruptly terminated due to serious illness on the part of the author.

What follows is a sample of how the author had planned to continue the series by building upon the instruction set foundation that had been laid in the early installments. Enjoy!

MACHINE LANGUAGE PROGRAMMING
THE SHARP PC-1500
AND RADIO SHACK PC-2
POCKET COMPUTERS

Once a truly interested ML fan acquires an understanding of the types of instructions that are available on a machine, he or she soon develops an itch to do some real ML programming. Let's satisfy that urge right now.

Clearing Memory

When batteries are installed in a PC, the individual bits in memory will "come up" in random states. Some set to the logic 1 state, some cleared to the 0 state. Sometimes it is not desirable to have areas of RAM in such a chaotic condition. One way for a ML programmer to set an area in memory to a known conditon is to fill it with zero bytes. That is, load bytes set to the value zero into a specific range of memory addresses. Sounds like a pretty simple procedure, right? It is! And it can be done in a whole lot of different ways.

Suppose, for example, that you wanted to clear out the section of memory normally used to store the fixed string variables A$ through D$. These are stored in memory addresses &78C0 - &78FF.

One way to accomplish this job would be to load CPU register A with the value zero. Next, the starting address of the area to be cleared might be set up in CPU register X. Once this had been done, STAI (X) directives could be used to stuff the contents of the accumulator (register A) into successive locations in memory as pointed to by the contents of CPU register X. Remember, the STAI (X) instruction automatically advances the value in X each time it is executed. There is just one more parameter to consider. How many times must the STAI (X) directive be repeated in order to clear out the desired block of memory?

Since the size of the memory block is known in this example, a counter could be established, say in register UL. Each time the STAI (X) command was performed, the count in UL could be decremented. When this count reached zero, it would be time to stop stuffing zeros into memory. Thus, one could use the sequence of directives (after the STAI (X) instruction) consisting of: DEUL and RBNZ #nn. That is, decrease the count in register UL and if it is still non-zero, then loop back to repeat the STAI (X) directive. If this method was used, in this example, then register UL would initially have to be set to the decimal count of 64 (hexadecimal 40), representing the number of bytes in memory (from &78C0 through &78FF) that were to be cleared. See the accompanying listing for a detailed example of this method.

If you were wide awake when you read the previous section of this series, then you might remember that this type of situation is ideal for the use of the BNZD #nn instruction. Since, however, the BNZD directive tests for zero *before* it decrements the contents in UL, then the count initially placed in UL must be one less than the number of loops (locations to be cleared). Thus, in this specific example, if BNZD was used, register UL would need to start out with a count of decimal 63 (which is hexadecimal &3F). An accompanying listing illustrates this method, too.

Yet another way of determining when to stop looping would be to check for an appropriate address value in CPU register X. In this case, when the value in X exceeded &78FF (i.e., reached &7900), then it would be time to discontinue the clearing operation. There are several ways this type of procedure might be implemented.

One way would be to set the ending value (say &78FF in this case) into another CPU register. CPU register Y would be available for such use in this example. A comparison could then be made between the contents of X and Y each time that X was advanced by the STAI directive. When the value in X exceeded that in Y, then it would be appropriate to stop the clearing operation. See the

Program *Routine for Clearing Memory Using RBNZ Instruction*

PG	LC	B1	B2	B3	B4	B5	LABELS	MNEMONICS	COMMENTS
00	00	85	00				CLRMM1	LDA #00	Load register A with zero.
00	02	48	78					LDXH #78	Set up register X to point to 16-bit address 678C0.
00	04	4A	C0					LDXL #C0	Set up register X to point to 16-bit address 678C0.
00	06	6A	40					LDUL #40	Set up counter (64 decimal here) in register UL.
00	08	41					CLRM1	STAI (X)	Stuff contents of A into memory, then increment pointer.
00	09	62						DEUL	Decrement the counter value in UL (UL=UL-1).
00	0A	99	04					RBNZ CLRM1	If counter is not zero, jump back to stuff another byte.

Program *Routine for Clearing Memory Using BNZD Instruction*

PG	LC	B1	B2	B3	B4	B5	LABELS	MNEMONICS	COMMENTS
00	00	85	00				CLRMM2	LDA #00	Load register A with zero.
00	02	48	78					LDXH #78	Set up register X to point to 16-bit address 678C0.
00	04	4A	C0					LDXL #C0	Set up register X to point to 16-bit address 678C0.
00	06	6A	3F					LDUL #3F	Set up counter (63 decimal here) to count-1 (64-1=63).
00	08	41					CLRM2	STAI (X)	Stuff zero byte into memory, advance memory pointer in X.
00	09	88	03					BNZD CLRM2	Check UL for zero, loop back if non-zero, (UL=UL-1).

Program *Routine for Clearing Memory Using Compare Operation to Terminate Loop*

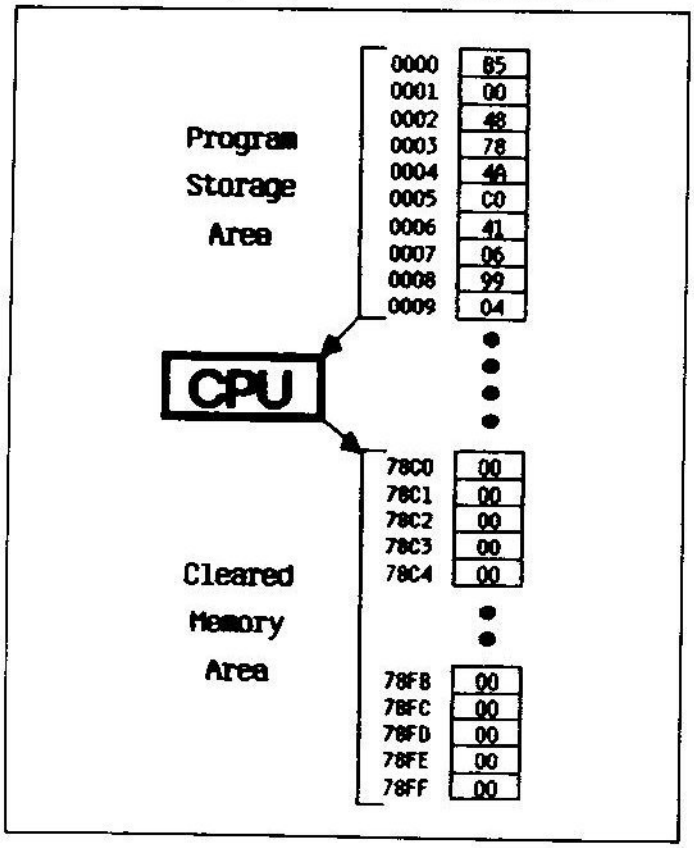
PG	LC	B1	B2	B3	B4	B5	LABELS	MNEMONICS	COMMENTS
00	00	85	00				CLRMM3	LDA #00	Load register A with zero.
00	02	48	78					LDXH #78	Set up register X to point to 16-bit address 678C0.
00	04	4A	C0					LDXL #C0	Set up register X to point to 16-bit address 678C0.
00	06	41					CLRM3	STAI (X)	Stuff zero byte into memory, advance memory pointer in X.
00	07	06						CPA XL	Check XL for zero (indicating address 67900 reached).
00	08	99	04					RBNZ CLRM3	Loop back to stuff next location if XL is not zero.

example program listing.

An even easier way, in this particular case, would merely be to test for the value in XL going to zero. That is because, when XL exceeds FF (because X reaches 78FF), it will go to the value 00 as X advances to the value 7900. That just happens to be the point at which we want to stop clearing memory in this particular example! A listing of this method is also provided for examination.

By now you should be convinced that ML programming is not an exact science. There are usually countless ways to approach a particular objective. Sometimes you can customize your approach depending on specific parameters. For instance, if you are trying to conserve your use of memory, you might try to devise a sequence of directives that uses a minimum amount of space. Or, you might be interested in maximizing speed of program execution. In that case, you could work towards selecting directives that could be performed in the minimum amount of time. (Do *not* make the assumption that the fastest program is the one with the fewest instructions! Various classes of instructions require different times to complete their execution. It may require a very detailed study in order to find a sequence of instructions that accomplishes an objective in a minimum amount of time!) In most practical applications, however, the actual sequence of instructions selected is not at all critical. Select or create a method you like and try it out!

Diagram *Memory Map of Clearing Operation.*



A Practical Memory Clearing Application

Have you ever developed a BASIC program that used a temporary variable array? That is, an array that you needed to continuously clear out in between operations with other arrays? If so, you know that you had to create a special program loop that would initialize all the elements in that specific temporary array. You could not use a BASIC command such as CLEAR as that would wipe out *all* of the other variables and array elements used in the program, something that we assume for this example, would not be desired.

Of course, there is nothing wrong with creating a BASIC program loop to clear out the elements in an array. It is just that, if the number of elements in the array is large, the process can take some time. If the clearing operation has to be done frequently, the amount of time devoted to this one aspect can become quite exasperating.

However, as a ML programmer, it is possible to devise a scheme to clear out the elements of an array in the proverbial "blink of an eye." Let us see how this could be done.

First, it is necessary to know a few facts about the operation of the BASIC interpreter provided in the PC-1500 (and Radio Shack PC-2). As you may be aware, the interpreter program stored on ROM organizes RAM memory in a specific fashion to serve its purposes. Thus, the lower range of available user memory is assigned for use by the PC's "softkeys" as the REServe memory area. Immediately above this area (in terms of memory addresses) is where user program statements for a BASIC program (or multiple programs) are stored. Finally, the "top" or highest address value locations in RAM are used for the storage of user-defined arrays and variables.

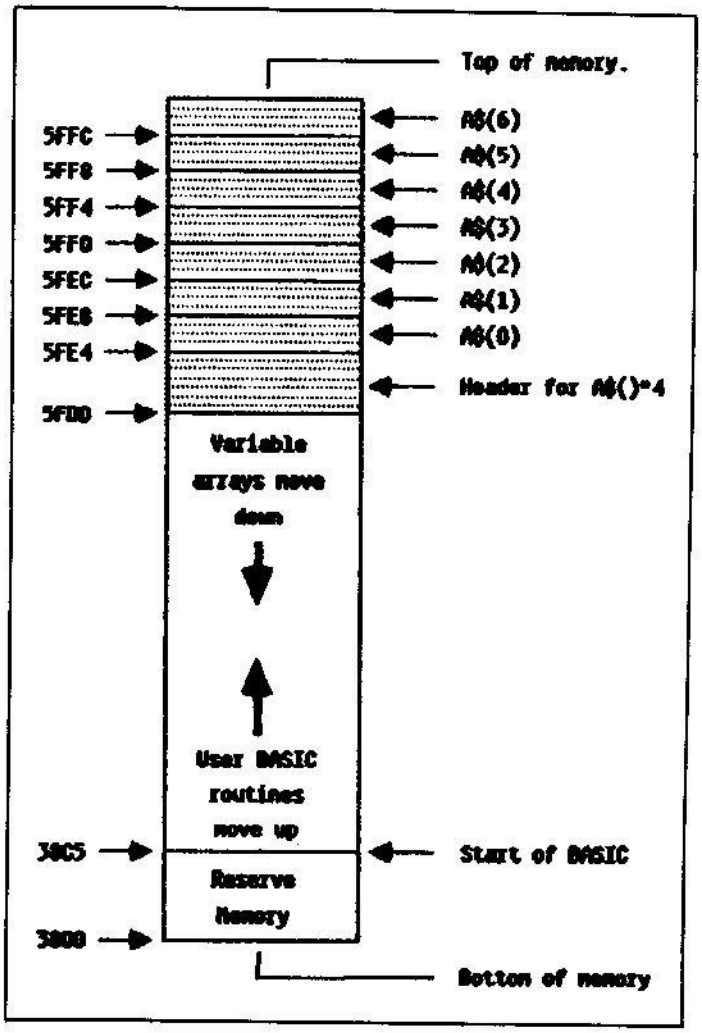
When the pocket computer is first turned on, the ROM program determines the bottom and top addresses of user RAM. The "page" (high order 8 bits) value of these locations are stored in the system RAM at addresses &7863 and &7864 respectively. This procedure is necessary as the range of user RAM can vary depending on which RAM expansion module (such as the CE-151, CE-155 or CE-161), if any, is installed in the PC. Thus, for example, if an 8K module was installed, location &7863 would contain &38 (which is the page portion of the address &3800 where RAM memory would begin). Location &7864 would contain &60. This is the page portion of the address &6000 which is one more than the top user RAM address (&5FFF) that is available in such a system.

(Knowing this information makes it possible to design a procedure that will automatically take account of the amount of memory in a user's system when it is used.)

When a BASIC programmer wants to create a variable array, it is necessary to issue a DIMension statement. This statement essentially blocks out space in user RAM for storage of the array elements. For purposes of illustration let us assume that the DIMension statement is the first statement contained in the user's BASIC program. Let us further assume that it is expressed as follows: DIM A$(6)*4. This means that a string array named A$() is being specified, that there are a total of 7 array elements (numbered 0 - 6) being reserved and that each element is to have room for 4 characters.

When the BASIC interpreter encounters this DIMension statement it will do the following: determine the top of memory (by examining system RAM location &7864), reserve 28 bytes of memory immediately below this location for storage of the array elements (7 elements with 4 characters reserved for each one), immediately below this it will record the array "header" information, (using 7 bytes of storage for this purpose). A summary of this process is shown in the accompanying diagram.

Diagram *Memory Map for Clearing an Array.*



The reason why it is necessary to assume that the DIMension statement is the first statement in the program in this description, is because any other arrays (or user-created variables) will be stored "beneath" this initial array. Thus, locating other arrays can become complicated and such complications are not needed at this point in the ML programming educational process. Right?

It will be worth knowing, so that you might customize such a routine to your own specific purposes, the following additional information about array elements: (1) The number of characters reserved for each element of a string array is precisely the number specified after the asterisk in the DIMension statement. If an asterisk is not used to specify such a value (which is limited to the range 1 to 80), then a value of 16 is assumed by the BASIC interpreter. (2) The number of characters reserved for each element of a numeric array is *always* eight! This is because all numeric values stored in such array elements are assumed to be in floating-point BCD format.

An Array-Clearing ML Routine

If we assume an array size of seven elements (numbered 0 through 6), with an element length of 4 characters, then we can calculate that the space needed by the array elements is exactly 28 bytes. (Note that this excludes the 7 bytes needed by the array "header." This is just as well, however, as we do not want to erase the contents of the array header!)

By examining the contents of memory location &7864, we can locate the start of memory. Actually, this location yields the first page address that lacks RAM, so user memory really begins on the highest possible address (&FF) on the next

Program *Routine for Clearing Memory that Terminates When Address Value Reached*

PC	LC	B1	B2	B3	B4	B5	LABELS	MNEMONICS	COMMENTS
00	00	48	78				CLRM4	LDXH #78	Set up register X to point to 16-bit address &78C0.
00	02	4A	C0					LDXL #C0	Set up register X to point to 16-bit address &78C0.
00	04	58	79					LDYH #79	Set up register Y to point to 16-bit address &7900.
00	06	5A	00					LDYL #00	Set up register Y to point to 16-bit address &7900.
00	08	B5	00				CLRM4	LDA #00	Load register A with zero.
00	0A	41						STAI (X)	Stuff zero into memory, advance memory pointer.
00	0B	94						LDA YH	Put contents of YH into the accumulator.
00	0C	86						CPA XH	Compare (YH-XH) to see if pointer page values are same.
00	0D	99	07					RBNZ CLRM4	Loop back if page values are not the same.
00	0F	14						LDA YL	Put contents of YL into the accumulator.
00	10	06						CPA XL	Compare (YL-XL) to see if pointer values are same.
00	11	99	08					RBNZ CLRM4	Loop back if page values are not the same.

Program *Routine for Clearing Array Elements.*

PG	LC	B1	B2	B3	B4	B5	LABELS	MNEMONICS	COMMENTS
71	50	58	78				CARRAY	LDYH #78	Set up register Y to point to 16-bit address &7864.
71	52	5A	64					LDYL #64	This is where top of memory value stored by ROM routines.
71	54	15						LDA (Y)	Fetch top of memory value into accumulator.
71	55	DF						DEA	Decrement page value by one to point to next lower page.
71	56	08						STA XH	Store top of memory (adjusted) page value in register XH.
71	57	4A	FF					LDXL #FF	Store low portion of top of memory address in XL.
71	59	6A	1C				CARRA1	LDUL #1C	Set up counter (28 decimal here) in register UL.
71	58	B5	00					LDA #00	Load the accumulator with zero.
71	5D	43					CARRA2	STAD (X)	Stuff accumulator into memory, then decrement pointer.
71	5E	62						DEUL	Decrement the counter value in UL (UL=UL-1).
71	5F	99	04					RBNZ CARRA2	If counter not zero, jump back to stuff another byte.
71	61	9A						RTS	Else, exit back to caller when counter equals zero.

lower page. Thus, if location &7864 contains the value &60 (indicating page &60 in the address &6000), the last location in RAM is at location &FF in the next lower page (&5F) or at address &5FFF. (This is precisely what would be encountered if a PC-1500 had an 8K RAM module such as the CE-155 installed.)

From there it is a simple matter to set up pointers and a byte counter within the CPU in order to erase the desired block of memory. However, now it will be appropriate to decrement the memory pointer as locations are cleared (instead of incrementing it as was done in previous routines). See the accompanying listing for the actual series of instructions that can accomplish this objective.

Check It Out With A Hybrid Program

You can test the operation of the array clearing routine by combining it with a BASIC program. The BASIC portion of the program can be used to DIMension the array, load the machine language routine into memory using POKE directives (since it is fairly short), and initialize the array with a set of known values.

Next, the initial values placed into the array can be displayed for checking purposes. The array-clearing ML routine can then be "called" using the CALL statement provided in BASIC. The contents of the array can then be displayed again to verify that the elements were properly cleared. This process may be repeated as long as desired for observational purposes. Here is the listing for such a *hybrid* program:

```
1000 "AA"DIM A$(6
     )*4
1010 GOSUB "CC"
1020 "BB"GOSUB "D
     D"
1030 GOSUB "EE"
1040 CALL &7150
1050 GOSUB "EE"
1060 GOTO "BB"
1090 END
1100 "CC"POKE &71
     50,&58,&78,&
     5A,&64,&15,&
     DF,8,&4A
1110 POKE &7158,&
     FF,&6A,&1C,&
     B5,0,&43,&62
     ,&99
1120 POKE &7160,4
     ,&9A
1130 RETURN
1200 "DD"FOR A=0
     TO 6
1210 A$(A)=STR$ (
     A)+STR$ (A)+
     STR$ (A)+
     STR$ (A)
1220 NEXT A
1230 RETURN
1500 "EE"WAIT 20
1510 FOR A=0TO 6
1520 PRINT A;"
      !";A$(A);"!
     ":PRINT " "
1530 NEXT A
1540 "FF"RETURN
```

(The nomenclature *hybrid* in this text refers to programs that combine BASIC and machine language methods within one general program, such as illustrated by this array-clearing example.)

Note that the ML routine is tucked into memory locations that are normally used for storing the string variables P$ and Q$. Keep this in mind if you intend to meld this array-clearing capability into some other program!